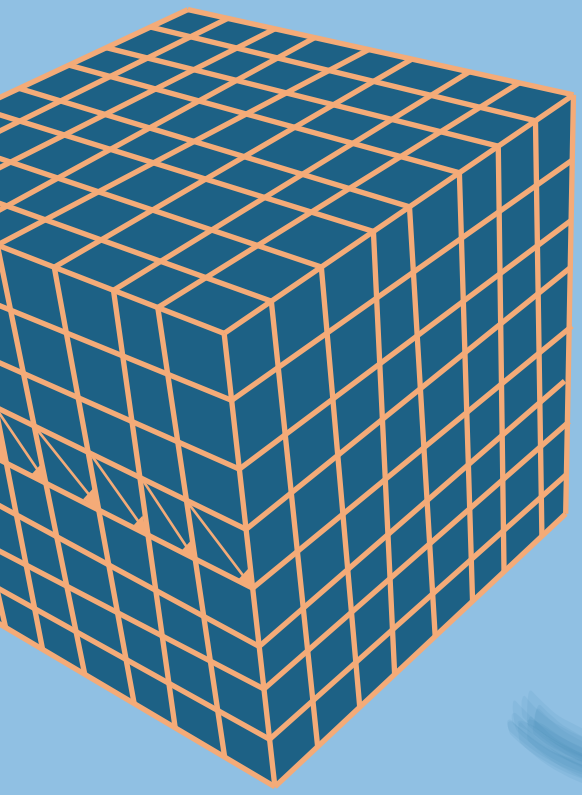
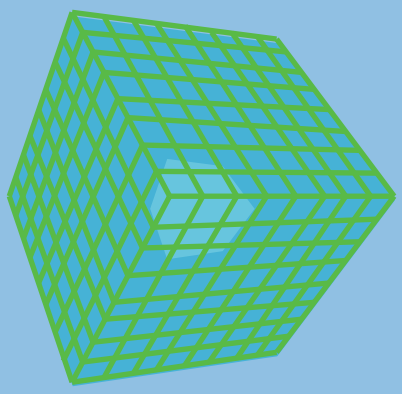
VERIFICATION OF PROGRAM PARALLELIZATION





SAEED DARABI

Verification of Program Parallelization

Saeed Darabi

Graduation Committee:

Chairman:	Prof.dr. J.N. Kok	University of Twente
Supervisor:	Prof.dr. M. Huisman	University of Twente
Members:	Prof.dr. P. Müller	ETH Zürich
	Prof.dr. G. Gopalakrishnan	University of Utah
	Dr.ir. A.L. Varbanescu	University of Amsterdam
	Prof.dr.ir. M.J.G. Bekooij	University of Twente
	Prof.dr.ir. A. Rensink	University of Twente

IDS Ph.D. Thesis Series No. 18-458
Institute on Digital Society
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede, The Netherlands

IPA Dissertation Series No. 2018-02
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

The work in this thesis is conducted within Correct and Efficient Accelerator Programming (CARP) project (287767), supported by European Commission.

ISBN: 978-90-365-4484-9
ISSN: 2589-4730 (IDS Ph.D. Thesis Series No. 18-458)
DOI: 10.3990/1.9789036544849
Available online at https://doi.org/10.3990/1.9789036544849

Typeset with LaTeX. Printed by GildePrint.
Cover design © by Annelien Dam

Verification of Program Parallelization

Dissertation

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof.dr. T.T.M. Palstra,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, 2 March, 2018 at 16:45 hrs.

by

Saeed Darabi

born on 21 September 1982
in Isfahan, Iran

This dissertation has been approved by:

Supervisor: Prof.dr. M. Huisman

To Elnaz and my parents

Acknowledgments

By writing these acknowledgments I am taking my last steps to finish this dissertation. I started four and half years ago the journey of this PhD research, like an ambitious mountaineer at the foothills of a mountain ridge, trading off which peak shall he aim at. After all the ups and downs, the joys and the hurts, now I am in my last steps on my way to the summit. These very moments are rich of great emotions: feeling proud of being able to accomplish it, feeling thankful to the people who supported me throughout this path and feeling confident to tackle the next peak in the mountain range of life.

Marieke and Jaco, I am grateful to both of you for giving me the opportunity to be part of the Formal Method & Tools (FMT) group. Working with such motivated and encouraging people was a great source of inspiration for me.

Marieke, thank you for being always supportive. You had always a clear vision about the main research objectives. Like the pole star, you helped me not to deviate from the important research goals throughout the path. At the same time you gave me sufficient freedom to pursue different research challenges and develop my own academic skills. Besides the scientific skills, working with you was also a great chance to develop my management skills in practice by observing you as a perfect example. It was always admirable for me how you manage so many different responsibilities in such a tight schedule.

Arend Rensink, thank you for the nice collaboration within the Advanced Logic course. It was an opportunity for me to learn more about the foundations of the techniques that I used in my thesis. My project teammates, Afshin Amighi, Stefan Blom, Wojciech Mostowski, Marina Zaharieva-Stojanovski, Wytse Oortwijn, and Freark van der Berg: thank you for all the technical discussions, collaborations and your helpful feedback; a special thank you to Stefan for our long and insightful discussions. My office mates, Tri, Lesley, Stefan, Wytse, and Freark, thanks for all the refreshing chats in the office and coffee corner.

I would like to thank all members of the committee for their willingness to

read and approve the thesis: Peter Müller, Ganesh Gopalakrishnan, Ana Lucia Varbanescu, Marco Bekooij and Arend Rensink. I am also grateful to European Commission, who funded this work via the CARP project.

All FMT members, together you have made a dynamic research atmosphere in the group; it is like a river: as soon as you are in, you have no way but to flow. I enjoyed the social events, lunch meetings, ice-skatings, outings, and all the other group activities. I and my wife are proud of being part of the FMT's prestigious running team. Stefano, thanks for inviting me infinitely often to sport activities: boxing, running and spinning. It was indeed a brain sport to find a new excuse every time.

Ida, Jeanette and Joke, I am sincerely grateful for the assistance provided by you. You have been always there to help me with all the official procedures. Joke, big thanks for your help on my very early days as a FMT member.

I am grateful to my family and my dear friends for their support and encouragement over the years. I thank my mother Tayebeh, for being my first teacher, and for igniting the passion for science in my mind; my father Hossein, for teaching me about diligence and endurance, throughout our mountaineering trips to Iran's highest peaks; and my brother Kaveh, for being always supportive and encouraging.

I saved the best part for my lovely wife. Elnaz, it is out of imagination what we have passed through together. Thank you for being so courageous. The best part of any success is the moment when I share it with you. Thank you for supporting me unconditionally over the years to achieve my dreams. We have made so many beautiful memories and many more are coming.

Finally, I would like to close these acknowledgements by two couplets from Rumi who influenced me more than other writers and philosophers.

چند بینی این و آن و نیک و بد بنگر آخر این و آن آمیخته
چند گویی بی نشان و با نشان بی نشان بین با نشان آمیخته

"Why are you so busy with this or that or good or bad, pay attention to how things blend.
Why talk about all the known and the unknown, see how the unknown merges into the known." Rumi

Abstract

This thesis presents techniques to improve *reliability* and prove *functional correctness* of parallel programs. These requirements are especially crucial in critical systems where system failures endanger human lives, cause substantial economic damages or security breaches. Today's critical systems are expected to deliver more and more complex and computationally intensive functions. In many cases these cannot be achieved without exploiting the computational power of multi- and even many-core processors via parallel programming. The use of parallelization in critical systems is especially challenging as on one hand, the non-deterministic nature of parallel programs makes them highly error-prone, while on the other hand high levels of reliability have to be guaranteed.

We tackle this challenge by proposing novel formal techniques for verification of parallel programs. We focus on the verification of data race freedom and functional correctness, i.e. a program behaves as it is expected. For this purpose, we use axiomatic reasoning techniques based on permission-based separation logic.

Among different parallel programming paradigms, in deterministic parallel programming, parallelization is expressed over a sequential program using high-level parallel programming constructs (e.g. parallel loops) or parallelization annotations; next the low-level parallel program (e.g. a multithreaded program or a GPGPU kernel) is generated by a parallelizing compiler.

First, we present a verification technique to reason about loop parallelizations. We introduce the notion of an iteration contract that specifies the memory locations being read or written by each iteration of the loop. The specifications can be extended with extra annotations that capture data-dependencies among the loop iterations. A correctly written iteration contract can be used to draw conclusions about the safety of a loop parallelization; it can also indicate where synchronization is needed in the parallel loop. Iteration contracts can be further extended to specify the functional behavior of the loop such that the functional correctness of the loop can be verified together with its parallelization safety.

Second, we propose a novel technique to reason about deterministic parallel programs. We first formally define the Parallel Programming Language (PPL), a simple core language that captures the main forms of deterministic parallel programs. This language distinguishes three kinds of basic blocks: parallel, vectorized and sequential blocks, which can be composed using three different composition operators: sequential, parallel and fusion composition. We show that it is sufficient to have contracts for the basic blocks to prove the correctness of the PPL program, and moreover that the functional correctness of the sequential program implies the correctness of the parallelized program. We formally prove the correctness of our approach. In addition, we define a widely-used subset of OpenMP that can be encoded into our core language, thus effectively enabling verification of OpenMP compiler directives, and we discuss automated tool support for this verification process.

Third, we propose a specification and verification technique to reason about data race freedom and functional correctness of GPGPU kernels that use atomic operations as a synchronization mechanism. We exploit the notion of resource invariant from Concurrent Separation Logic to specify the behaviour of atomic operations. To capture the GPGPU memory model, we adapt this notion of resource invariant such that *group resource invariants* capture the behaviour of atomic operations that access locations in local memory, which are accessible only to the threads in the same work group, while *kernel resource invariants* capture the behaviour of atomic operations that access locations in global memory, which are accessible to all threads in all work groups. We show the soundness of our approach and we demonstrate the application of the technique in our toolset.

This thesis presents a set of verification techniques based on permission-based separation logic to reason about the data race freedom and functional correctness of program parallelizations. Our reasoning techniques address different forms of high-level and low-level parallelization. For high-level parallel programs, we first formalize the main features of deterministic parallel programming in PPL and discuss how PPL programs and consequently, real-world deterministic parallel programs (e.g. OpenMP programs) are verified. For low-level parallel programs, we specifically focus on reasoning about GPGPU kernels. At the end we discuss how the presented verification techniques are chained together to reason about the semantical equivalence of high-level parallel programs where they are automatically transformed into low-level

parallel programs by a parallelizing compiler. Thus, effectively enabling a holistic verification solution for such parallelization frameworks.

Contents

List of Figures

CHAPTER 1

INTRODUCTION

"The more I think about language, the more it amazes me that people ever understand each other at all."

– Kurt Gödel

This thesis presents techniques to improve *reliability* and prove *functional correctness* of parallel programs. Software reliability is the probability of failure-free software operation for a specific period of time in a specific environment [Mus80, L+96]. Functional correctness of software means that it behaves as defined by the functional requirements of the system. Reliability and correctness are two important design criteria in almost any system, but they are especially crucial requirements in the development of *critical systems*.

Critical systems are failure-sensitive systems. Depending on the consequences of a failure they are classified into different categories of safety, mission, business, and security critical systems. In general critical systems are those systems whose failure may endanger human lives, damage the environment, cause substantial economic loss, disrupt infrastructure, or results in information leakage [Kni02]. Traditional critical systems are spacecrafts, satellites, commercial airplanes, defense systems, nuclear power plants and weapons. However, if we look carefully, critical systems are much more pervasive nowadays and this is an increasing trend. In medicine: heart pacemakers, infusion pumps, radiation therapy machines, robotic surgery, in critical infrastructures: water systems, electrical generation and distribution systems, emergency services such as the 112 line, transportation control and banking are some of the modern examples of critical systems on which we are depending more and more.

Traditionally critical systems have been designed using only mechanical and electronic parts where reliability can be achieved by setting high *safety factor* or by adding redundant parts to the design. For example a standard elevator design requires an "eight safety factor". This means that it can carry eight times more load than the expected load for which it is designed. This is a large safety margin and a waste of resources but it is paid to ensure the right level of reliability.

Critical systems have evolved over time to deliver more complex functions

and at the same time be more programmable. Throughout this evolution, software components and parallel programming were introduced to the designs of these systems and then gradually became an essential part of them [SEU+15]. Although software-based systems are easily programmable such that a new functionality can be implemented in a couple of hours, ensuring the absolute safety of the system is in fact multiple orders of magnitude harder and even impossible in some cases; this lesson has only been learnt through several deadly and disastrous software failures [LT93, Dow97, JM97]. It is also well understood that software testing is not an ultimate solution as "*testing can only show the presence of bugs; not their absence*" [Dij72].

Unlike mechanical and electronic parts, software does not age, wear-out or rust. So all software-related errors are in principle *design faults*. They are human mistakes; the failure of designers and developers to understand the system and its operational environment, to communicate unambiguously, and to predict the effect of actions and changes when they propagate in the system.

If we track these faults down in the process of system development. We realize that most of them are caused by inconsistent assumptions or lack of precise information. In practice, the requirements of the system are ambiguous in the early stages of the design; even clients do not know clearly what precisely their expected product should be. So software development proceeds by making a lot of assumptions. These assumptions if documented, they are written in natural language that is not sufficiently precise to be used for detecting the possible inconsistencies. Later, developers build up the system on top of those imprecise or potentially inconsistent assumptions. Consequently the absolute reliability and functional correctness of the system become unverifiable.

Human beings are not evolved to be precise, only an approximated perception of the world would suffice for our survival [HSP15]. However, as a creator of a software system, even a small amount of deviation and imprecision may eventually cause huge errors. This is perhaps the main reason that our current science and engineering is so dependent on mathematics. Logic from its earliest forms of Aristotle's syllogism [ari17] to later well-defined forms of mathematical logic known as *formal logic* is one of the oldest branches of mathematics. Formal logic provides a mathematical method to specify knowledge in an unambiguous way. When it is employed in software design and development, formal logic can be used as a powerful instrument to precisely specify a software system and its components. Only under a mathematically precise specification of a

software system its absolute reliability and functional correctness is provable. According to John MacCarthy, "it is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last" [McC61].

The interaction of formal logic and computer science has been so much profound so far that the formal logic has been called "the calculus of computer science". We shortly highlight the parts of this interaction which are specifically related to this thesis and refer to Reference [HHI+01] for further readings. The first application of formal logic in reasoning about the correctness of computer programs dates back to the seminal works of Floyd and Hoare [Flo93, HW73] where Hoare logic is introduced for the first time. The logic enables us to prove the functional correctness of sequential programs assuming that they terminate. In a significant breakthrough, Raynolds, O'Hearn, Yang and others [Rey02, ORY01, IO01] extended Hoare logic to *separation logic* based on the Burstall's observation [Bur72] that separate program texts which work on separate sections of the store can be reasoned about independently. Next O'Hearn developed the concurrent variant of separation logic [O'H04, O'H07, O'H08], *Concurrent Separation Logic* (CSL) that enables modular reasoning about the reliability and correctness of concurrent programs. To support concurrent reads, Bornat and others present *Permission-Based Separation Logic* (PBSL) [BCOP05] that combines CSL with Boyland's fractional permissions [Boy03]. We discuss the course of this evolution with more details in Chapter 2.

This thesis contributes to the above-mentioned research line by developing verification methods based on permission-based separation logic to reason about the safety and functional correctness of parallel programs. We also facilitate the practical applicability of this verification approach by prototyping the developed techniques in a verification toolset.

1.1. Concurrency, Parallelism and Data Races

According to Moore's law the number of transistors on a chip doubles every two years [Moo98]. Independent of how the law will survive in the next decades, it played an influential role on hardware development and also the way we program that hardware in the past half a century. With more transistors on a chip, customers expect more storage and faster processing. However, in a single-core processor increasing the processing speed, achieves at the cost of higher

power dissipation and more complex processor design. To manage the power consumption and complexity, processor vendors has recently favored multi-core processor designs. That, however has not really resolved the complexity issue but rather lifted it up to the software level; as exploiting the full computing power of the modern multi-core architectures requires software developers to give up the simpler and more reliable way of sequential programming in favor of parallel programming.

Writing a parallel program typically starts with decomposing the expected functionality into a set of smaller tasks (also known as threads or processes). Tasks are executed concurrently on a single or multi-core processor. When inter-task communication is necessary, developer can weave a synchronization protocol into the program that restrict the interleaving of statements such that the deterministic execution of the program is guaranteed. If the synchronization is correct, the visible outcome of the concurrent execution of the tasks should be as if they are executing sequentially by a single processor. In fact this programming method has been invented and being used even before the emergence of multi-core processors, in the form of *concurrent programming*.

Concurrency and parallelism are often used as synonyms in the literature, as well as in this thesis. However, there is a slight distinction between the two concepts that we would like to elaborate on. Concurrency is about making a model of software in which the expected functionality is distributed among smaller tasks, how the tasks are communicating and how data is shared among them but concurrency does not propose how the tasks are executed on a specific hardware platform (e.g. on a single or multi-core or even many-core processor). The actual binding of the concurrency model to a specific hardware platform is called parallelism. So concurrency defines a software model while parallelism represents how that model is bound to a specific hardware platform. Despite this subtle difference, because of simplicity we consider these two terms synonym in this thesis.

Concurrent programming is known to be difficult and error-prone [Lee06]. What makes concurrency specifically difficult is when processes need to communicate. Concurrent execution of independent processes is safe; however in many applications inter-process communications are inevitable. One efficient way for inter-process communication is to use shared memory. Processes can read and write from the shared memory and even to or from the same location. So one process can write a value which may later read by another

process. The problem is that there is no guarantee that the read of the reader process happens after the write of the writer process (assuming that this is the expected behaviour); unless they are properly synchronized. If two accesses are read, there is no execution order which yields a harmful result. However, the uncontrolled race of two processes where they access to a same location and at least one of the accesses is a write access is called *data race*. This is the source of many errors in concurrent programs that eventually leads to non-deterministic functional behaviour and unreliability.

A closer look into the way concurrency is used, especially in scientific and business applications, reveals that concurrency often is not intrinsic to the function of the system but rather it is an optimization step. Therefore in many applications it is possible to express the full functionality of the system by a sequential program and the more efficient parallel version of the program can later be generated by a *parallelizing compiler*. This is a *high-level parallelization* approach that allows the parallelization to be defined over a sequential program. As this parallelization method only produces the parallel programs that are able to represent the deterministic functional behaviour of a sequential program (namely their original sequential counterpart), this approach is often called *deterministic parallel programming*. One problem is that standard sequential programming languages are not sufficiently expressive to describe parallelization. Thus in many cases the compiler cannot decide if for example a loop is parallelizable or not.

The high-level description of parallelization over sequential programs can be implemented differently. One approach is to use annotations in the form of compiler directives to hint the compiler where and how to parallelize [ope17d, ope17b, BBC+15]. OpenMP [ope17d] is one of the popular language extensions for shared memory parallel programming that follows this approach. The approach has also inspired projects such as CARP [CAR17] which are aimed at increasing the programmability and portability of programs written for many-core processors by hiding the complexity of hardware dependent low-level details and presenting the parallelization in a high-level language (i.e. PENCIL [BBC+15, BCG+15]). The downside of the approach is that the compiler fully relies on user annotations. Thus, an incorrect parallelization annotation leads to a racy low-level parallel program. Other approaches are diverse in the range between extending sequential programming languages with parallelization constructs (e.g. parallel loop construct) and wrapping parallelization into some

high-level library functions [GM12, BAAS09, Rob12, Par17].

Given a high-level parallel program, the parallelizing compiler generates the low-level parallel program for a specific target platform. It can be a multi-threaded C program to be executed on a multi-core or a single core processor or an OpenCL kernel to be executed on a many-core accelerator processor such as a GPU. As the final product in this programming method is the low-level parallel program, it is important to show that the generated low-level parallel program is indeed *semantically equivalent* to its high-level counterpart (i.e. it behaves same as its high-level counterpart). For example the functional behaviour of an OpenMP program is preserved when it is translated into an OpenCL kernel and runs on a GPU.

This thesis tackles the challenge of ensuring reliability and functional correctness of the high-level approach to program parallelization discussed above. We specifically use permission-based separation logic to specify and verify: (1) the high-level parallelization annotations are used correctly, (2) the high-level program respects its functional properties and (3) the low-level translation preserves the same functionality and it is data race free. The next section elaborates these verification challenges.

1.2. Verification Challenges

This section briefly presents the main challenges that we study in this thesis. Each of the following challenges is addressed by one of the chapters of this thesis: Chapter 3 addresses Challenge 1, Chapter 4 and Chapter 5 discuss how we tackle Challenge 2 and 3 respectively; and Chapter 6 presents our solution to Challenge 4.

- *Challenge 1: How to specify and verify loop parallelization?*

  The iterative structure of loops makes them suitable for parallelization. However, only the loops that have data-independent iterations can be safely parallelized. In the presence of loop-carried dependencies, parallelization is either not possible or it can be done only if proper inter-iteration synchronizations are used. The challenge is how to verify that a loop which is claimed parallel by the developer is indeed parallelizable. Moreover, we want to be able to verify if the loop can be parallelized by adding extra synchronizations.

- *Challenge 2: How to precisely formalize high-level deterministic parallel programming?*

  We discussed how deterministic parallel programming simplifies parallelization by expressing parallelism over a sequential program; so the low-level parallel program can be automatically generated. This approach makes parallelization more accessible, productive, platform independent and maintainable. Although the approach is commonly used especially in high performance scientific and business applications, it is not properly formalized. This hinders the application of formal approaches to reason about the correctness of this parallel programming paradigm. The challenge is how to formalize a core parallel language that captures the main features of deterministic parallel programming such that it can be used for static analysis and verification of real-world deterministic parallel programs.

- *Challenge 3: How to reason about the functional correctness and data race freedom of high-level parallel programs?*

  Given a formalized core language for deterministic parallel programming, the challenge is how to verify functional correctness and data race freedom of the programs written in it. As writing manual specifications is costly and time consuming, can we reduce specification overhead by automating some parts of the reasoning?

- *Challenge 4: How to show that low-level parallel programs, in particular GPGPU programs, are functionally correct and data race free?*

  Among the available techniques for low-level parallel programming (e.g. using POSIX threads), General Purpose GPU (GPGPU) programming is a rather new and rapidly growing paradigm. There are only limited static analysis techniques that specifically deal with data race freedom of GPGPU programs [BCD+12, LG10, BHM14]. Among them, the work by Blom et al. [BHM14] presents a deductive verification approach based on permission-based separation logic that enables reasoning about both functional correctness and data race freedom of GPGPU programs. However, the method is limited to GPGPU programs that do not use atomic operations. The challenge is to extend the technique such that the GPGPU programs that use both barriers and atomic operations can be verified as well. Additionally, when a GPGPU program is automatically generated

from a high-level parallel program, how can the semantic equivalence of the GPGPU program and its high-level counterpart be ensured?

1.3. Contributions

This thesis contributes with novel techniques for reasoning about functional correctness and data race freedom of parallel programs. We list the following contributions:

- A specification and verification technique for reasoning about the safety and functional correctness of loop parallelizations;
- Formalizing a simple Parallel Programming Language (PPL), which captures the main forms of deterministic parallel programs;
- A verification technique for reasoning about the data race freedom and functional correctness of PPL programs;
- An algorithm for encoding OpenMP programs into PPL that enables verification of OpenMP programs;
- A specification and verification technique that adapts the notion of resource invariants to the GPU memory model and enables us to reason about the data race freedom and functional correctness of GPGPU kernels containing atomic operations;
- Demonstrating the practical applicability of the proposed verification techniques by prototyping them in our VerCors toolset.

1.4. Outline

The thesis is organized as follows:

Chapter 1 (Introduction).

Chapter 2 (Permission-based Separation Logic): presents background on separation logic and how it can be used for reasoning about concurrent programs. It also gives a high-level overview on the architecture of our VerCors toolset.

Chapter 3 (Verification of Parallel Loops): introduces the notion of iteration contracts and employs it to reason about loop parallelization. This chapter is based on the papers "Verification of Loop Parallelisations" and "Verifying Parallel Loops with Separation Logic", which were published at FASE 2015 [BDH] and PLACES 2014 [BDH14] respectively.

Chapter 4 (Parallel Programming Language): explains the syntax and operational semantics of our parallel programming language. It also gives an introduction to OpenMP and discusses how OpenMP programs are translated into PPL programs. This chapter is based on the paper "A Verification Technique for Deterministic Parallel Programs", which was published at NFM 2017 [DBH17a] and its extended version [DBH17b].

Chapter 5 (Verification of Deterministic Parallel Programs): discusses our verification approach to reason about deterministic parallel programs represented in PPL. This chapter is based on the paper "A Verification Technique for Deterministic Parallel Programs", which was published at NFM 2017 [DBH17a].

Chapter 6 (Verification of GPGPU Programs): gives a short introduction to GPGPU programming and presents how we reason about the data race freedom and functional correctness of GPGPU kernels that use atomic operations. This chapter is based on the paper "Specification and Verification of Atomic Operations in GPGPU Programs", which was published at SEFM 2015 [ADBH15].

Chapter 7 (Conclusion): concludes the thesis and identifies some promising new directions for future work.

CHAPTER 2

PERMISSION-BASED SEPARATION LOGIC

"The job of formal methods is to elucidate the assumptions upon which formal correctness depends."

– Tony Hoare

The verification techniques that we discuss in this thesis are built on top of Permission-Based Separation Logic (PBSL) [Boy03, BCOP05, Hur09b, HHHA14]. Separation Logic [Rey02] is an extension of Hoare Logic [Hoa69], originally proposed to reason about imperative pointer-manipulating programs. In this thesis we use permission-based separation logic as the basis of our specification language to reason about program parallelizations. Section 2.1 first gives an overview on the main concepts of the logic. Then Section 2.2 formally presents the syntax and semantics of the formulas. Finally in Section 2.3, we give a high-level overview about how reasoning with permission-based separation logic for parallel programs is implemented in our VerCors toolset. Later in Chapters 3, 5 and 6, we elaborate on how the proposed verification technique in each chapter is implemented as part of the VerCors toolset.

2.1. The Basic Concepts of Separation Logic

Hoare logic is a formal system for reasoning about program correctness. In this approach, a program or part of it, is specified by pre- and postconditions. A precondition is a predicate that formally describes the condition that a program or part of it relies upon for a correct execution. The postcondition is the predicate that specifies the condition a program establishes after its correct execution. When they are used for specifying program components (e.g. functions) pre- and postcondition present a mathematically precise *contract* for that component. For a function they are a contract between the implementation of the function and the caller of the function (the client). The precondition of the function should be fulfilled by the client and in return the client relies on the postcondition of the function after the call to the function.

A program is *partially correct* with respect to its specification if, assuming the precondition is true just before the program executes, then if the program

terminates normally without exceptions, the postcondition is true. The program is *totally correct* if it is partially correct and also the termination of the program is guaranteed.

To reason about program correctness, Hoare logic uses *Hoare triples*. A Hoare triple when it is used to specify the partial correctness is of the form:

$$\{P\}S\{Q\}$$

where P and Q are pre- and postconditions respectively and S is the statement(s) of the program. The total correctness meaning of a Hoare triple, specified with the Hoare triple $[P]S[Q]$, is that if the execution of S starts in a state where P is true, then S will terminate in a state where Q is true.

Consider the Hoare triple $\{x == 1\}x = x * 10\{x > 5\}$. The triple is correct because given $x == 1$, if we multiply x by 10, the result is $x == 10$, which implies $x > 5$. So given the precondition, the postcondition is satisfiable after the execution of the statement $x = x * 10$. However the given postcondition can be strengthened if it is substituted by $x == 10$. The latter postcondition is the *strongest postcondition* for the provided statement.

Given the specifications in the form of Hoare triples the partial correctness of programs is deduced using the axioms and rules of Hoare logic. For a detailed discussion on how the rules and axioms of Hoare logic are used to prove program correctness, we refer to [Apt81, Apt83, Hoa69, HW73, LGH+78].

Hoare logic presents a formal system to prove the correctness of imperative programs. However, the approach is not effectively applicable to some programming techniques. A well-known class of programs of this kind is pointer-based programs that are widely used in many application domains. The problem with these programs is that they allow a shared mutable data structure to be referenced from more than one point in the program. So a memory location might be altered by a seemingly unrelated expression. This can happen when there are at least two pointers that are aliases (i.e. refer to the same location); in this way they are essentially different pointer variables referring to the same memory location. A number of solutions have been proposed for reasoning about pointer-based programs (a partial bibliography is given in Reference [IO01]). Among them separation logic has gained widespread popularity. The other approaches either have a limited applicability or they are extremely complex. Although separation logic was initially presented as a solution for reasoning about the correctness of pointer-based programs,

it turned out that the logic has a great potential to be extended to reason about concurrent program. Before showing how separation logic is used for verification of concurrent programs, we first discuss its main building blocks.

The central idea in separation logic is to specify program properties over disjoint (separated) partitions of the memory. Therefore, instead of finding a complex predicate which is valid globally in the program, one can specify valid properties on smaller and disjoint parts of the memory. This is enabled by the separating conjunction operator $\star$. The predicate $P \star Q$ asserts that P is a valid assertion on the memory partition m_1 and Q is another valid assertion on another memory partition m_2, and m_1 and m_2 are disjoint (i.e. there is no memory location which belongs both to m_1 and m_2).

An important basic predicate in separation logic is the *points-to* predicate $x \mapsto v$, meaning that x points to a location on the memory, and this location contains the value v. The points-to predicates can be conjoined by separating conjunction $\star$ operator. For example, $x \mapsto 1 \star y \mapsto 2$ asserts that there exists (only) two separate memory cells pointed to by pointer variables x and y where the x location contains the value 1 and the y location contains the value 2. The presence or absence of the term "only" in the previous sentence distinguishes between two main flavors of separation logic in the literature: *intuitionistic separation logic* [IO01] and *classical separation logic* [JP11, BCO05]. If the term is present, it adds an extra semantics that the memory only contains two memory cells with the specified properties, the classical flavor, while when the term is absent, it leaves open the possibility of having or not having other memory cells besides the specified ones, the intuitionistic flavor. In this thesis we use the intuitionistic flavor of the logic.

According to the definition of the separating conjunction, the predicate $x \mapsto \_ \star x \mapsto \_$ is a contradiction as both the left-hand and right-hand operand of the separating conjunction are referring to the same memory cell; so they cannot be disjoint. Note that the assertion $x \mapsto 2 \wedge y \mapsto 2$ either states that there are two memory cells pointed by x and y pointers where both contain the value 2 or there is one memory cell referenced to by both pointers x and y and it has the value 2. Thus, x and y are aliased.

Thus, separation logic enables the specification of properties over separated partitions of the memory. This separation is exploited by the logic's proof rules to enable an interesting aspect of the logic, so-called *local reasoning*: the parts of the program which access to disjoint memory partitions can be reasoned about

independently. One of the important proof rules, behind the local reasoning in separation logic is the *frame rule*:

$$\frac{\{P\}S\{Q\} \quad (\mathit{modifies}\ S \cap \mathit{vars}\ R = \emptyset)}{\{P \star R\}S\{Q \star R\}}\ [\textbf{Frame Rule}]$$

This rule expresses that if we can prove $\{P\}S\{Q\}$ locally on a memory partition, we can conclude that $\{P \star R\}S\{Q \star R\}$ holds for an extended memory. This means that S has not modified anything in the extended part of the memory R. The side-condition means that the free variables in R have also not modified by S.

In addition to reasoning about pointer-based programs, separation logic can also be used to reason about concurrent programs: if two threads only access to disjoint parts of the memory, they do not interfere and thus can be verified in isolation. This means that the correctness proof of a concurrent program can be decomposed into the correctness proof of its threads, given the fact that the threads are accessing to disjoint memory locations. This is formulated in the parallel rule [O'H07, O'H08, Vaf11]:

$$\frac{\{P_1\}S_1\{Q_1\} \quad \cdots \quad \{P_n\}S_n\{Q_n\} \quad (C_1\ \mathit{and}\ C_2)}{\{P_1 \star \cdots \star P_n\}S_1 || \cdots || S_n\{Q_1 \star \cdots \star Q_n\}}\ [\textbf{Parallel Rule}]$$

The rule explains that if the predicates P_1 to P_n hold on n separate memory partitions, and we distribute each partition to a separate thread, then we can reason about each thread in isolation, and finally combine their postconditions $Q_1 \star \cdots \star Q_n$. The rule has two side-conditions, C_1 states that a variable that is changed by one thread cannot appear in another thread unless it is owned by that thread and C_2 states that the thread S_i must not modify variables that are free in P_j or Q_j where $i \neq j$. However, as Bornat showed the side-conditions of both parallel and frame rule can be removed by treating variables as resources [BCY06].

Separation logic provides a modular way to reason about concurrent programs. However, the logic only allows reasoning about threads that operate on disjoint locations; so simultaneous reads from the same location by different threads is not allowed. To address this issue the logic has been extended with the notion of fractional permissions to denote the right to either read from or write to a location [Boy03, BCOP05, BH14, BDHO17, Hur09b, HHHA14, Vip17]. Any fraction in the interval $(0, 1)$ denotes a *read permission*, while 1 denotes a *write permission*. Permissions can be split and combined, but soundness of the

logic prevents the sum of the permissions for a location over all threads to exceed 1. This means that at most one thread at a time can hold a write permission, while multiple threads can simultaneously hold a read permission to a location. This guarantees that if permission specifications can be verified, the program is data race free. The set of permissions that a thread holds are often called its *resources*.

These (ownership) fractions are often denoted as $\mathsf{Perm}(e, \pi)$ indicating that a thread holds an access right π to the memory location e, where any fraction of π in the interval $(0, 1)$ denotes a read permission and 1 denotes a write permission. Write permissions can be split into read permissions, while multiple read permissions can be combined into a write permission. For example, $\mathsf{Perm}(x, 1/2) \star \mathsf{Perm}(y, 1/2)$ indicates that a thread holds read permissions to access the disjoint locations x and y. If a thread holds $\mathsf{Perm}(x, 1/2) \star \mathsf{Perm}(x, 1/2)$, this can be merged into a write permission $\mathsf{Perm}(x, 1)$. Equivalently, the write permission can be split into read permissions; for example when a master thread shares a piece of memory between the threads it forks.

In this thesis we use the VerCors version of permission-based separation logic to reason about the correctness of program parallelization. In different chapters, we present specification techniques to reason about different classes of parallel programs. The syntax and semantics of the separation logic formulas used in our specifications are defined formally in the next section.

2.2. Syntax and Semantics of Formulas

Our specification language combines separation logic formulas with the Java Modeling Language (JML). In this way we exploit the expressiveness and readability of JML while enabling the use of separation logic for reasoning about data race freedom and functional correctness. JML annotations that are used in the examples of this thesis are standard and commonly known by programmers. We discuss them later where they are actually used in our examples. To learn more about JML, we refer to [LBR99, LBR06]. In this section we explain the syntax and semantics of the separation logic formulas that in combination with JML annotations construct our specification language.

Formulas F in our logic are built from first-order logic formulas b, permission predicates $\mathsf{Perm}(e_1, e_2)$, conditional expressions $(\cdot ? \cdot : \cdot)$, separating conjunction $\star$, and universal separating conjunction $\bigstar$ over a finite set I. The

syntax of formulas is formally defined as follows:

$$F ::= b \mid \mathsf{Perm}(e_1, e_2) \mid b?F : F \mid F \star F \mid \bigstar_{i \in I} F(i)$$
$$b ::= \textbf{true} \mid \textbf{false} \mid e_1 == e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2 \mid \ldots$$
$$e ::= v \mid n \mid [e] \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots$$

where b is a side-effect free boolean expression, e is a side-effect free arithmetic expression, [.] is a unary dereferencing operator, thus $[e]$ returns the value stored in the address e in shared memory, v ranges over the variables and n ranges over numerals. We assume the first argument of the $\mathsf{Perm}(e_1, e_2)$ predicate is always an address and the second argument is a fraction. We use the array notation $a[e]$ as syntactic sugar for $[a + e]$ where a is a variable containing the base address of the array a and e is the subscript expression; together they point to the address $a + e$ in shared memory.

Our semantics mixes concepts of Implicit Dynamic Frames [SJP12] and separation logic with fractional permissions. In this respect it is different from the traditional separation logic semantics and more aligned towards the way separation logic is implemented over traditional first order logic tooling. For further reading on the relationship of these two semantics we refer to the work of Parkinson and Summers [PS11].

To define the semantics of formulas, we assume the existence of the following domains: Loc, the set of memory locations, VarName, the set of variable names, Val, the set of all values, which includes the memory locations, and Frac, the set of fractions ($[0, 1]$).

We define memory as a map from locations to values $h : \mathsf{Loc} \rightarrow \mathsf{Val}$. A memory mask is a map from locations to fractions $\pi : \mathsf{Loc} \rightarrow \mathsf{Frac}$ with unit element $\pi_0 : l \mapsto 0$ with respect to the point-wise addition of heap masks. A store is a function from variable names to values: $\sigma : \mathsf{VarName} \rightarrow \mathsf{Val}$.

Formulas can access the memory directly; the fractional permissions to access the memory are provided by the Perm predicate. Moreover, a strict form of self-framing is enforced. This means that the boolean formulas expressing the functional properties in pre- and postconditions and also in invariants should be framed by sufficient resources (i.e. there should be sufficient permission fractions for the memory locations that are accessed by the boolean formula).

The semantics of expressions depends on a store, a memory, and a memory mask and yields a value: $\sigma, h, \pi \, [e\rangle \, v$. The store σ and the memory h are used to determine the value v, the memory mask π is used to determine if the expression is correctly framed. This means that there is sufficient permission

for the memory locations that are required to be accessed for the evaluation of the expression. For example, the rule for array access is:

$$\frac{\sigma, h, \pi \,[e\rangle\, i \qquad \pi(\sigma(a) + i) > 0}{\sigma, h, \pi \,[a[e]\rangle\, h(\sigma(a) + i)}$$

where $\sigma(a)$ is the initial address of array a in the memory and i is the array index that is the result of evaluation of index expression e. Apart from the check for correct framing as explained above, the evaluation of expressions is standard and we do not explain it any further.

The semantics of a formula, given in Figure 2.1, depends on a store, a memory, and a memory mask and yields a memory mask: $\sigma, h, \pi \,[F\rangle\, \pi'$. The given mask π represents the permissions by which the formula F is framed. The yielded mask π' represents the additional permissions provided by the formula. Hence, a boolean expression is valid if it is true and yields no additional permissions, (rule **Boolean**), while evaluating a Perm predicate yields additional permissions to the location, provided the expressions are properly framed (rule **Permission**). We overload standard addition $+$, summation Σ, and comparison operators to be respectively used as pointwise addition, summation and comparison over the memory masks. These operators are used in the rules **SepConj** and **USepConj**. In the rule **SepConj**, each formula F_1 and F_2 yield a separate memory mask, π' and π'' respectively, where the final memory mask is calculated by pointwise addition of two memory masks, $\pi' + \pi''$. The rule checks if $F1$ is framed by π and F_2 is framed by $\pi + \pi'$. The rule **USepConj** extends the similar evaluation by quantifying over a set of formulas conjoined by the universal separating conjunction operator. Note that the permission fraction on any location in the memory cannot exceed one, this is checked in the rules **USepConj** and **Permission**.

Finally, a formula F is valid for a given store σ, memory h and memory mask π if starting with the empty memory mask π_0, the required memory mask of F is less than π:

$$\sigma, h, \pi \models F, \text{ if } (\sigma, h, \pi_0 \,[F\rangle\, \pi') \wedge (\pi' \leq \pi)$$

$$\frac{\sigma, h, \pi \, [b\rangle \, \mathsf{true}}{\sigma, h, \pi \, [b\rangle \, \pi_0} \, [\textbf{Boolean}]$$

$$\frac{\sigma, h, \pi \, [e_1\rangle \, l \qquad \sigma, h, \pi \, [e_2\rangle \, f \qquad \pi(l) + f \leq 1}{\sigma, h, \pi \, [\mathsf{Perm}(e_1, e_2)\rangle \, \pi_0[l \mapsto f]} \, [\textbf{Permission}]$$

$$\frac{\sigma, h, \pi \, [b\rangle \, \mathsf{true} \qquad \sigma, h, \pi \, [F_1\rangle \, \pi'}{\sigma, h, \pi \, [b?F_1 : F_2\rangle \, \pi'} \, [\textbf{Cond 1}] \quad \frac{\sigma, h, \pi \, [b\rangle \, \mathsf{false} \qquad \sigma, h, \pi \, [F_2\rangle \, \pi'}{\sigma, h, \pi \, [b?F_1 : F_2\rangle \, \pi'} \, [\textbf{Cond 2}]$$

$$\frac{\sigma, h, \pi \, [F_1\rangle \, \pi' \qquad \sigma, h, \pi + \pi' \, [F_2\rangle \, \pi''}{\sigma, h, \pi \, [F_1 \star F_2\rangle \, \pi' + \pi''} \, [\textbf{SepConj}]$$

$$\frac{\forall i \in I : \sigma, h, \pi \, [F(i)\rangle \, \pi_i \quad \pi + \Sigma_{i \in I} \pi_i \leq 1}{\sigma, h, \pi \, [\bigstar_{i \in I} F(i)\rangle \, \Sigma_{i \in I} \pi_i} \, [\textbf{USepConj}]$$

Figure 2.1: Semantics of formulas in permission-based separation logic

2.3. VerCors Toolset

To demonstrate the practical applicability of the verification techniques developed in this thesis, we implement them as part of our VerCors toolset. This section briefly describes the high-level architecture of the toolset. Later in each chapter, when it is necessary, we provide more information about the implementation details. The open source distribution of the toolset is available at [Ver17b].

The VerCors toolset was originally developed to reason about multi-threaded Java programs. However, it has been extended to support the verification of OpenCL kernels [BHM14] and a subset of OpenMP for C programs. The toolset leverages the existing verification technology as discussed in this thesis: it encodes programs via several program transformation steps into *Viper* programs [JKM+14]. Viper is an intermediate language for separation logic-

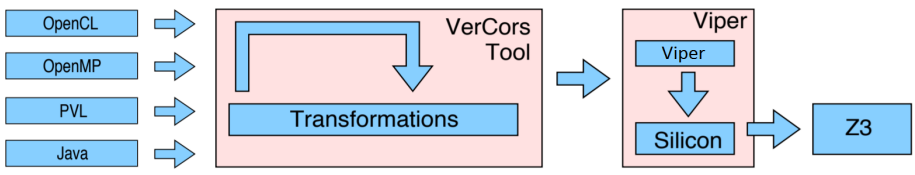


Figure 2.2: VerCors toolset overall architecture

like specifications, used by the Viper (Verification Infrastructure for Permission-based Reasoning) toolset [JKM+14, Vip17]. Viper programs can be verified in the Silicon verifier [JKM+14, HKMS13, MSS16] that uses the Z3 SMT solver [DMB08] to discharge logical queries. Figure 2.2 sketches the overall architecture of our VerCors toolset.

CHAPTER 3

VERIFICATION OF PARALLEL LOOPS

"Begin with the simplest examples."

– David Hilbert

Parallelizing compilers aim to detect loops that can be executed in parallel. However, this detection is not perfect. Therefore developers can typically add compiler directives to declare that a loop is parallelizable. Any loop annotated with such a compiler directive is assumed to be a parallel loop by the compiler. In this method, a developer's failure in providing correct compiler directives misleads the compiler and results in a racy parallelization.

In this chapter we discuss how to verify that loops that are declared parallel by a developer can indeed safely be parallelized. This is achieved by adding specifications to the program that when verified guarantee that the program can be parallelized without changing its behaviour. Our specifications stem from permission-based separation logic as discussed in Chapter 2.

Concretely, for each loop body we add an *iteration contract*, which specifies the iteration's resources (i.e. the variables read and written by one iteration of the loop). We show that if the iteration contract can be proven correct without any further annotations, the iterations are independent and the loop is parallelizable.

For loops with loop-carried data dependencies, we can add additional annotations to capture the dependencies. These annotations specify how resources are transferred from one iteration to another iteration of the loop. We then identify the class of annotation patterns for which we can prove that the loop can be vectorized because they capture forward loop-carried dependency.

We also discuss that how iteration contracts can be easily extended to capture the functional behaviour of the loop. This allows to seamlessly verify the functional correctness of the parallel loop together with its parallelizability.

Our approach is motivated by our work on the CARP project [CAR17]. The project aims at increasing the programmability of accelerator processors such as GPUs by hiding their low-level hardware complexity from the developers.

The content of this chapter is based on the following publications of the author: "Verification of loop parallelization" [BDH] and "Verifying Parallel Loops with Separation Logic" [BDH14].

As part of the project the PENCIL programming language has been developed [BBC+15]. PENCIL is a high-level programming language. Its core is a subset of sequential C which is extended with parallelization annotations. The annotations are used by developers to hint to the compiler which loops are parallelizable and how they can be parallelized. Our verification technique is originally developed to reason about PENCIL's parallelization annotations. However the technique is directly applicable to other programming languages or libraries that have similar loop parallelization constructs, such as `omp parallel for` in OpenMP [ope17d], parallel_for in C++ TBB [Thr] and Parallel.For in .NET TPL [Micb].

As another part of the CARP project, a parallelizing compiler has been developed [VCJC+13] to take the PENCIL programs as input and generate low-level GPU kernels. Later in Section 6.6 we discuss how the verified iteration contract, including the specifications of functional properties, can be translated into a verifiable contract for the generated low-level program, more specifically the generated GPU kernel. As we discuss extensively in Chapter 6, the produced kernel contract can be used to verify the generated kernel and to prove its functional equivalence to its parallel loop counterpart.

The main contributions of this chapter are the following:

- a specification technique, using iteration contracts and dedicated permission transfer annotations that can capture loop dependencies;
- a soundness proof that loops respecting specific patterns of iteration contracts can be either parallelized or vectorized; and
- tool support that demonstrates how the technique can be used in practice to verify parallel loops.

Outline. The remainder of this chapter is organized as follows. After some background information on loop dependencies and loop parallelization, Section 3.2 discusses how iteration contracts are used to specify parallel loops and how the extra resource transfer annotations captures loop-carried data dependencies. Section 3.3 explains the program logic that we use to verify parallel loops. The soundness of our approach is proven in Section 3.4. To demonstrate the practical usability of the proposed technique, in Section 3.5 we describe how we implement the technique in our VerCors toolset. Finally we

end this chapter by discussing related work in section 3.6 and conclusion and some directions for future work in section 3.7.

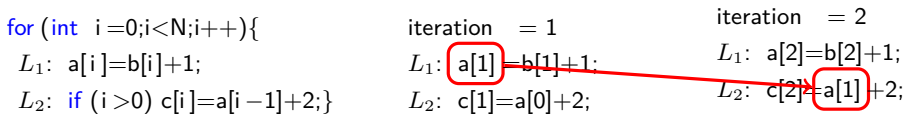
3.1. Loop Parallelization

We provide some background on loop parallelization and we discuss how different kinds of loop-carried data dependencies corresponds to different loop-level parallelizations.

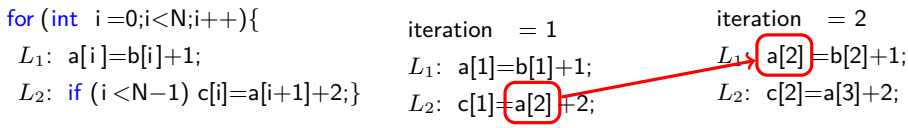
Loop-carried Dependencies. Given a loop, there exists a *loop-carried dependence* from statement S_{src} to statement S_{sink} in the loop body if there exist two iterations *i* and *j* of that loop, such that: (1) $i < j$, and (2) instance *i* of S_{src} and instance *j* of S_{sink} access the same memory location, and (3) at least one of these accesses is a write. The *distance* of a dependence is defined as the difference between *j* and *i*. We distinguish between *forward* and *backward* loop-carried dependencies. When S_{src} syntactically appears before S_{sink} there is a *forward loop-carried dependence*. When S_{sink} syntactically appears before S_{src} (or if they are the same statement) there is a *backward loop-carried dependence*.

Example 3.1 (Loop-carried Dependence)**.** *The examples below show two different types of loop-carried dependence. In (a) the loop has a* forward loop-carried dependence, *where L_1 is the source and L_2 is the sink, as illustrated by unrolling iteration 1 and 2 of the loop. In general, the i^{th} element of the array a is shared between iteration i and $i-1$. In (b) the loop has a* backward loop-carried dependence, *because the sink of the dependence (L_1) appears before the source of dependence (L_2) in the loop body.*

(a) An example of forward loop-carried dependence



(b) An example of backward loop-carried dependence



Typically the loops are parallelized by assigning a thread to handle one iteration or a chunk of loop's iterations. In this way loops with no loop-carried data dependences, called *independent loops* in this thesis, are *embarrassingly parallelizable*. However, the parallelization of loops with loop-carried data dependencies depends on the type of dependencies. More specifically, for loops with forward loop-carried dependencies, parallelization is safely possible if the sequential execution of data-dependent statements is preserved. This means that, in the part (a) of the example above, statement L_2 in iteration i always executes after statement L_1 in iteration $i-1$ no matter which thread interleaving is chosen. In practice this can be implemented either by using appropriate synchronizations (e.g. by inserting a barrier between statement L_1 and L_2) or by *vectorization* of the loop. The statements of a vectorized loop are executed based on Single Instruction Multiple Data (SIMD) execution model where the safe execution order of data-dependent statements is preserved by definition. Later in Section 6.6, we discuss how the loop with forward loop-carried dependence in part (a) of the example above, is executed in the SIMD fashion by being translated into a GPU kernel.

3.2. Specification of Parallel Loops with Iteration Contracts

This section first introduces the notion of *iteration contract* and explains how it captures different forms of loop-carried data dependence.

3.2.1 Specification of Loop-carried Data Dependencies

The classical way to specify the behaviour of a loop is by means of an invariant that has to be preserved by every iteration of the loop. However, loop invariants offer no insight into possible parallel executions of the loop. Instead we consider every iteration of the loop in isolation. Each iteration is specified by its iteration contract, such that the precondition of the iteration contract specifies the resources that a particular iteration needs, and the postcondition specifies the resources that become available after the execution of the iteration. In other words, we treat each iteration as a specified block [Heh05]. For convenience, we present our technique on non-nested for-loops with K statements that are executed during N iterations; however, our technique can be generalized to

Listing 1 Iteration contract for an independent loop

```
for (int i=0; i < N; i++)
/*@
    requires Perm(a[i], 1) ** Perm(b[i],1/2);
    ensures  Perm(a[i], 1) ** Perm(b[i],1/2);
@*/
{ a[i]= 2 * b[i];}
```

nested loops as well. Each statement S_k labeled by the label L_k consists of an atomic instruction I_k, which is executed if a guard g_k is true, i.e., we consider loops of the following form:

for (int j=0; j $<$ N; j++){ $body(j)$ }

where $body(j)$ is L_1: if (g_1) I_1; ... L_K: if (g_K) I_K;

There are two extra restrictions. First, the iteration variable j cannot be assigned anywhere in the loop body. Second, the guards must be expressions that are constant with respect to the execution of the loop body, i.e., they may not contain any variable that is assigned within the iteration.

Listing 1 shows an example of an independent loop extended with its iteration contract. This contract requires that at the start of the iteration i, permission to write a[i] is available, as well as permissions to read b[i]. Further, the contract ensures that these permissions are returned at the end of the iteration i. The iteration contract implicitly requires that the *separating conjunction of all iteration preconditions* holds before the first iteration of the loop, and that the *separating conjunction of all iteration postconditions* holds after the last iteration of the loop. For example, the contract in Listing 1 implicitly specifies that upon entering the loop, permission to write the first N elements of a must be available, as well as permission to read the first N elements of b.

To specify *dependent loops*, we need to specify what happens when the computations have to *synchronize* due to a dependence. During such a synchronization, permissions should be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. To specify such a transfer we introduce two annotations: send and recv:

//@ L_S: if ($g_S(j)$) { send $\phi(j)$ to L_R, d; }
//@ L_R: if ($g_R(j)$) { recv $\psi(j)$ from L_S, d; }

A send specifies that at label L_S the permissions and properties denoted by

Listing 2 Iteration contracts for loops with loop-carried dependences

(a)

```
for (int i=0; i < N; i++) /*@
    requires Perm(a[i], 1) ** Perm(b[i],1/2) ** Perm(c[i], 1);
    ensures  Perm(b[i],1/2) ** Perm(a[i],1/2) ** Perm(c[i], 1);
    ensures  i>0 ==> Perm(a[i-1],1/2);
    ensures  i==N-1 ==> Perm(a[i],1/2);  @*/
{
    a[i]=b[i]+1;
    //@ L1:if (i< N-1) send Perm(a[i],1/2) to L2,1;
    //@ L2:if (i>0) recv Perm(a[i-1],1/2) from L1,1;
    if (i>0) c[i]=a[i-1]+2;
}
```

(b)

```
for (int i=0; i < N; i++) /*@
        requires Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i], 1);
        requires i==0 ==> Perm(a[i],1/2);
        requires i < N-1 ==> Perm(a[i+1],1/2);
        ensures  Perm(a[i], 1) ** Perm(b[i],1/2) ** Perm(c[i], 1);   @*/
{
   //@ L1:if (i>0) recv Perm(a[i],1/2) from L2,1;
   a[i]=b[i]+1;
   if (i < N-1) c[i]=a[i+1]+2;
   //@ L2:if (i < N-1) send Perm(a[i+1],1/2) to L1,1;
}
```

formula ϕ are transferred to the statement labeled L_R in iteration $i + d$, where i is the current iteration and d is the distance of dependence. A recv specifies that permissions and properties as specified by formula ψ are received. In practice, the information provided by either send or recv statement is sufficient for the inference of the other annotation. So to reduce the annotation overhead, only one of them can optionally be provided by the developer. However, in this chapter to make the specifications more readable, we write the loop specifications completely using the both send and recv annotations.

The send and recv annotations can be used to specify loops with both forward and backward loop-carried dependencies. Listing 2 shows the specified instances of the code in Example 3.1. These examples are verified in the VerCors

toolset [Ver17a].

We discuss the annotations for part (a) in some detail. Each iteration i starts with a write permission on a[i] and c[i], and a read permission ($\frac{1}{2}$) on b[i]. The first statement is a write to a[i], which needs write permission. The value written is computed from b[i], for which a read permission is needed. The second statement reads a[i−1], which is not allowed unless read permission is available. This statement is not executed in the first iteration, because of the condition $i > 0$. For all subsequent iterations, permission must be obtained. Hence a send annotation is specified before the second assignment that transfers a read permission on a[i] to the next iteration (and in addition, keeps a read permission itself). The postcondition of the iteration contract reflects this: it ensures that the original permission on c[i] is released, as well as the read permission on a[i], which was not sent, and also the read permission on a[i-1], which was received. Finally, since the last iteration cannot transfer a read permission on a[i], the postcondition of the iteration contract also specifies that the last iteration returns this non-transferred read permission on a[i].

The send annotations indicate an order in which the iterations have to be executed, and thus how the loop can be parallelized. Any execution that respects this order yields the same behaviour as the sequential execution of the loop. For the forward dependence example, this means that it can be vectorized, i.e. we have to add appropriate synchronization to the parallel program to ensure permissions can be transferred as they are specified or the loop body can be executed in lock-step fashion using SIMD instructions. However, for the backward dependence example, only sequential execution respects the ordering; meaning that the loop is inherently sequential and cannot be parallelized. This is because of the fact that in the presence of backward loop-carried dependence the computations in the iteration i use the result produced in one of the previous iterations (e.g. the iteration $i - 1$ in the example).

Generalization for Nested Loops. As mentioned before, the technique can be applied to nested loops as well. For a nested loop, each loop is specified with its own iteration contract. Assume the loop LP_1 has n nesting levels (i.e. there are $n - 1$ loops inside LP_1 nested one into the other) and each loop has an iteration variable $i_1, \ldots, i_n$ respectively. Using our method each loop is specified with its own iteration contract. The iteration contract of LP_n, the inner-most loop, is parameterized by the iteration variable of LP_n (i.e. i_n) while the iteration

Listing 3 Extended iteration contract with the specification of functional properties

```
for (int i=0; i<N; i++) /*@
  requires Perm(a[i], 1) ** Perm(b[i],1/2) ** Perm(c[i], 1);
  requires b[i]==i;
  ensures  Perm(a[i],1/2) ** Perm(b[i],1/2) ** Perm(c[i], 1);
  ensures  i>0 ==> Perm(a[i-1],1/2);
  ensures  i==N-1 ==> Perm(a[i],1/2);
  ensures  a[i]==i+1 && b[i]==i && (i>0 ==> c[i]==i+2); @*/
{
  a[ i ] = b[ i ] + 1;
  //@ L1: if (i < N-1) send Perm(a[i],1/2) ** a[i]==i+1 to L2, 1;
  //@ L2: if (i>0) recv Perm(a[i-1],1/2) ** a[i-1] ==i from L1, 1;
  if (i>0) c[ i ] = a[ i-1 ] + 2;
}
```

contract of LP_{n-1} is parameterized by the iteration variable of LP_{n-1} (i.e. i_{n-1}) and is quantified over i_n. This continues, up to the iteration contract of the outer-most loop LP_1 that is parameterized by its own iteration variable i_1 and quantified over the iteration variables of all inner loops (i.e. $i_2, \ldots, i_n$). The data-dependencies that can appear in different nesting levels, are captured by send/recv annotations as discussed. Note that the application of our technique to nested loops requires the statement labeling method to be defined such that all statement instances at different nesting levels become uniquely identifiable.

3.2.2 Specification of Functional Properties

In addition to the specification of loop-carried data dependencies, iteration contracts can be extended to specify the functional behaviour of loops. This allows us to combine the reasoning about the parallelizability of loops with the reasoning about their functional correctness. To do this, the pre- and postcondition of iteration contracts are extended with first order logic formulas. They specify the state of the shared memory just before starting an iteration of the loop and the state of shared memory at the end of the execution of that iteration. Thus they capture the contribution of each iteration of the loop. At the end, the universal conjunction over pre- and postcondtion of all iterations specifies the functional behaviour of the loop.

As an example Listing 3 shows the extended iteration contract of the loop with forward-loop carried dependence in Listing 2. Note that to prove the postcondition c[i]==i+2 the prover needs to know about the value of a[i −1] that is determined only after the execution of a[i]=b[i]+1 in the previous iteration and transferred to the next iteration through the send/recv channel.

3.3. Verification of Iteration Contracts

This section discusses how the parallelizbability of a loop can be verified by checking the loop against its iteration contract. To prove the correctness of an iteration contract, we propose appropriate program logic rules. As mentioned above, an iteration contract implicitly gives rise to a contract for the loop. The following rule says that if the iteration contract is correct for any execution of the loop body then this contract is true:

$$\frac{\{P(j)\}\ body(j)\ \{Q(j)\} \quad \forall j.j \in [0..N)}{\{\bigstar_{j=0}^{N-1} P(j)\}\ \mathsf{for\,(int\ \ j=0;j<N;j++)\{}\ body(j)\ \mathsf{\}}\ \{\bigstar_{j=0}^{N-1} Q(j)\}}\ [\textbf{ParLoop}]$$

Note that this rule for a loop with an iteration contract is a special case of the rule for parallel execution, which allows arbitrary blocks of code to be executed in parallel (see e.g. [O'H07]).

The rules for the send and recv are similar in spirit to the rules that are typically used for permission transfer upon lock releasing and acquiring, (see e.g. [HHH08]). In particular, send is used to give up resources that the recv acquires. This is captured by the following two proof rules:

$$\frac{}{\{P\}\ \mathsf{send}\ P\ \mathsf{to}\ L, d\ \{\mathsf{true}\}}\ [\textbf{send}] \qquad \frac{}{\{\mathsf{true}\}\ \mathsf{recv}\ P\ \mathsf{from}\ L, d\ \{P\}}\ [\textbf{recv}] \tag{3.1}$$

Receiving permissions and properties that were not sent is unsound. Therefore, send and recv annotations have to be properly matched, meaning that:

(i) if S_r is the statement $\mathsf{if}\ (g_r(j))\ \ \mathsf{recv}\ \psi(j)\ \mathsf{from}\ L_s, d;$ then S_s is the statement $\mathsf{if}\ (g_s(j))\ \ \mathsf{send}\ \ \phi(j)\ \mathsf{to}\ L_r, d;$

(ii) if the recv is enabled in iteration j, then d iterations earlier, the send should be enabled, i.e.,

$$\forall j \in [0..N).g_r(j) \implies j \geq d \wedge g_s(j-d) \tag{3.2}$$

(iii) the information and resources received should be implied by those sent:

$$\forall j \in [d..N).\phi(j-d) \implies \psi(j) \tag{3.3}$$

In other words, the rules in Equation 3.1 cannot be used unless the syntactic criterion (i) and the proof obligations (ii) and (iii) hold.

3.4. Soundness

Next, we show that a correct iteration contract capturing a loop independence or a forward loop-carried dependence indeed implies that a loop can be parallelized or vectorized, while the behaviour of its sequential counterpart is preserved.

To construct the proof, we first define the semantics of the three loop execution paradigms: sequential, vectorized, and parallel. We also define the instrumented semantics for a loop specified with an iteration contract. Next, to prove the soundness of our approach we show that the instrumented semantics of an independent loop is equivalent to the parallel execution of the loop, while the instrumented semantics of a loop with a forward dependence is an extension of the vectorized execution of the loop. The functional equivalence of the two semantics is shown by transforming the executions in one semantics into the executions in the other semantics by swapping adjacent independent execution steps.

3.4.1 Semantics of Loop Executions

To keep our formalization tractable, we split the loop semantics into two layers. The upper layer determines which sequences of atomic statements, called *executions*, a loop can have. The lower layer defines the effect of each atomic statement, and we do not discuss this further here, as it is standard.

As above, we develop our formalization for non-nested loops with K guarded statements. We instantiate the loop body for each iteration of the loop, so we have (L_i^j: if(g_i^j) I_i^j;) as the instantiation of the i^{th} statement in the j^{th} iteration of the loop. We refer to this instance of statements as S_i^j. The semantics of a statement instance $[\![S_i^j]\!]$ is defined as the atomic execution of the instruction I_i^j labeled by L_i^j provided its guard condition g_i^j holds, otherwise it behaves as a skip. If we execute iterations one by one in sequential order and we preserve the

program order of the loop body, we will have a sequence of statement instances starting from S_1^0 and ending at S_K^{N-1}. Intuitively this is the semantics of the sequential execution of the loop.

Definition 3.1. *An* execution *c is a finite sequence $t_1, t_2, ..., t_m$ of statement instances such that t_1 is executed first, then t_2 is executed and so on until the last statement t_m.*

To define the set of executions describing the parallel and vectorized semantics of a loop, we define auxiliary operators *concatenation* and *interleaving* over the sets of executions. We define two versions of concatenation, plain concatenation ($++$) and *synchronized concatenation* ($\#$), which prevents data races between statements by inserting a barrier b that acts as a memory fence:

$$\begin{aligned} C_1 \mathbin{++} C_2 &= \{c_1 \cdot c_2 \mid c_1 \in C_1, c_2 \in C_2\} \\ C_1 \mathbin{\#} C_2 &= \{c_1 \cdot b \cdot c_2 \mid c_1 \in C_1, c_2 \in C_2\} \end{aligned}$$

We lift concatenation to multiple sets as follows:

$$\begin{aligned} \mathsf{Concat}_{i=1}^{N} C_i &= C_1 \mathbin{++} \cdots \mathbin{++} C_N \\ \mathsf{SyncConcat}_{i=1}^{N} C_i &= C_1 \mathbin{\#} \cdots \mathbin{\#} C_N \end{aligned}$$

Next, interleaving defines how to weave several executions into a single execution. This is parametrized by a *happens-before* order $<$, in order not to violate restrictions imposed by the program semantics. To define the interleaving operator ($\mathsf{Interleave}_<$), we use an auxiliary operator that denotes interleaving with a fixed first step of the i^{th} execution: ($\mathsf{Interleave}_<^i$):

$$\mathsf{Interleave}_<^{i=1..N} c_i = \mathsf{Interleave}_<(c_1, \cdots, c_N) = \bigcup_{i=1}^{N} \mathsf{Interleave}_<^i(c_1, \cdots, c_N)$$

$$\mathsf{Interleave}_<^i(\epsilon, \cdots, \epsilon) = \{\epsilon\}$$

$$\mathsf{Interleave}_<^i(c_1, \cdots, c_{i-1}, \epsilon, c_{i+1}, \cdots, c_N) = \emptyset \quad \text{, if } \exists c_{j \neq i} \neq \epsilon$$

$$\mathsf{Interleave}_<^i(c_1, \cdots, s_1.c_i', \cdots, c_N) =$$

$$\{s_1 \cdot x \mid x \in \mathsf{Interleave}_<(c_1, \cdots, c_i', \cdots, c_N), \nexists s \in x.s < s_1\} \quad \text{, otherwise}$$

where ϵ is the empty execution and s_1 is the first statement of c_i. In each step s_1 is extended with all possible interleaving of the other executions including the remaining statements of c_i, c_i'. This recursively generates all the interleavings which start with the statements of c_i, the set $\mathsf{Interleave}_<^i(c_1, \cdots, c_i, \cdots, c_N)$. Note that extending the statements of c_i is allowed only if the extension does not violate the happens-before order $<$. We overload the membership operator $\in$ to be used also for sequences such that $s \in x$ means that the statement s is in the sequence x.

We use two happens-before orders: *program order* (PO), which maintains the order of statements as if they are executed sequentially and *specification order* (SO), which extends the program order by enforcing that for every matching pair of send and recv, the send statement happens-before the recv statement. Both orders maintain the order between a barrier and the statements preceding and following it.

Now we are ready to define the semantics of the different loop executions. A *sequential execution* executes all loop iterations sequentially, *parallel execution* allows any interleaving that preserves program order within the loop body and *vectorized execution* runs multiple iterations in lock-step.

Definition 3.2. *Suppose we have a loop LP in standard form.*

- *Its* sequential execution semantics *is* $[\![LP]\!]^{Seq} = \mathsf{Concat}_{j=0}^{N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$
- *Its* parallel execution semantics *is* $[\![LP]\!]^{Par} = \mathsf{Interleave}_{\mathsf{PO}}^{j=0..N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$
- *Its* vectorized execution semantics *for vector length V is*

$$[\![LP]\!]^{Vec(V)} = \mathsf{Concat}_{v=0}^{(N/V)-1}\mathsf{SyncConcat}_{i=1}^{K}\left(\mathsf{Interleave}_{\emptyset}^{j=vV..vV+V-1}[\![S_i^j]\!]\right)$$

We define the *instrumented semantics* to capture the behaviour of *LP* in the presence of its specifications. This semantics contains all possible executions respecting the specification order (SO). It is formalized by parameterizing the interleaving operator with SO.

Definition 3.3. *The instrumented semantics of a loop LP is*

$$[\![LP]\!]^{Spec} = \mathsf{Interleave}_{\mathsf{SO}}^{j=0..N-1}\mathsf{Concat}_{i=1}^{K}[\![S_i^j]\!]$$

3.4.2 Correctness of Parallel Loops

In the previous section, we defined the semantics of parallelized and vectorized executions in terms of possible traces of atomic steps. This section proves, under certain conditions, that each of those traces is data race free and yields the same result as the sequential execution yields. Equivalence is established by considering traces modulo reordering independent steps and while ignoring steps that make no modifications. First, we formally define these notions. Then we present our correctness theorems.

To determine if two steps are independent and/or can cause a data race, we need to know for every atomic step, t, which locations in memory it writes,

$\mathsf{write}(t)$, which locations in memory it reads, $\mathsf{read}(t)$, and by which thread it is executed. We define the set of accessed locations as $\mathsf{access}(t) = \mathsf{write}(t) \cup \mathsf{read}(t)$. Now we define a *data race* in a trace as a pair of statements that both access a location, where at least one access is a write, and are not ordered by the happens-before relation:

Definition 3.4. *An execution contains a* data race *with respect to a happens-before order $<$, if it contains two steps s and t, such that* $\mathsf{write}(s) \cap \mathsf{access}(t) \neq \emptyset \wedge \neg(s < t \vee t < s)$.

To reason about different execution orders, the equivalence of executions is defined in terms of swapping the order of steps which are not in the happens-before relation. The following proposition states that this does not change the end result of a data race free execution.

Proposition 3.1. *In a data race free execution, swapping two adjacent statements which are unordered in the happens-before relation does not change the behaviour of that execution.*

Proof. Because the statements are unordered and the execution is data race free, the set of locations written by one of the actions cannot affect the set of locations accessed by the other. Hence neither step can see the effect of the other. □

The traces in the different semantics do not just differ by their order, but also by steps that are used to enforce synchronization. To compare the functional result of two threads, we only look at the steps in those traces that actually modify locations that are relevant to the program semantics.

Definition 3.5. *Given two executions c_1 and c_2. The executions c_1 and c_2 are* functionally equivalent *if* $\mathsf{mods}(c_1) = \mathsf{mods}(c_2)$, *where*

$$\mathsf{mods}(c) = \begin{cases} \epsilon & \text{, if } c = \epsilon \\ \mathsf{mods}(c') & \text{, if } c = t \cdot c' \wedge \mathsf{write}(t) = \emptyset \\ t \cdot \mathsf{mods}(c') & \text{, if } c = t \cdot c' \wedge \mathsf{write}(t) \neq \emptyset \end{cases}$$

The correctness of the various loop semantics depends on the correctness of the instrumented semantics:

Theorem 3.1. *Given a loop LP with a valid specification.*

1. *All executions in $[\![LP]\!]^{Spec}$ are data race free.*

2. *All executions in* $[\![LP]\!]^{Spec}$ *and* $[\![LP]\!]^{Seq}$ *are functionally equivalent.*

Proof. 1. Because there is a valid specification, all invariants of permission-based separation logic hold. In particular, for every location the sum over the permissions held by all threads for that location cannot exceed 1. Suppose that a statement s occurs before t where one writes a location l and the other accesses it and they are not ordered by happens-before ($s \not< t$). Assume that s needs a fraction p permission on l and t needs a fraction q on the same location. Because s does not necessarily happens-before t, they might be executed by two different threads implying that $p + q \leq 1$. This contradicts the assumption as either $p = 1$ and $q > 0$ or vice versa.

2. We prove that every execution in $[\![LP]\!]^{Spec}$ is functionally equivalent to the single execution $[\![LP]\!]^{Seq}$, by showing that any execution can be reordered until it is the sequential execution using Proposition 3.1.
Assume that the first n steps of the given execution are in the same order as the sequential execution. Then step t_{n+1} in the sequential execution has to be somewhere in the given sequence. Because each sequence contains the same steps and the sequential execution is in happens-before order, all of the steps that have to happen before t_{n+1} are already included in the $t_1, \cdots, t_n$ prefix. Therefore data race freedom of all executions in $[\![LP]\!]^{Spec}$, proved in the first part, only implies that the step t_{n+1} is independent of all the steps after the prefix and before itself in the given sequence. Thus t_{n+1} can be swapped with its prefix statements one-by-one until it is the next step in the given sequence. We then repeat until the whole sequence matches with the sequential order.

□

The correctness of the parallelization of independent loops is an immediate corollary of this theorem.

Corollary 3.1. *Given a loop with a valid specification, which does not make use of* send *or* recv.

1. *All executions in* $[\![LP]\!]^{Par}$ *are data race free.*

2. *All executions in* $[\![LP]\!]^{Par}$ *and* $[\![LP]\!]^{Seq}$ *are functionally equivalent.*

Proof. Because the specification does not make use of send or recv for each execution in $[\![LP]\!]^{Par}$ there is a corresponding execution in $[\![LP]\!]^{Spec}$, as the program order coincides with the specification order. The validity of specification and Theorem 3.1 imply that all executions in $[\![LP]\!]^{Spec}$ are data race free and functionally equivalent to the single sequential execution $[\![LP]\!]^{Seq}$ and thus so for $[\![LP]\!]^{Par}$.

□

This proof is straightforward because in this case, the program order and synchronization order coincide, thus the set of parallel executions is equivalent to the set of instrumented executions. However, if the specifications use send and recv then some parallel execution order may contain data races. But if the send occurs before the matching recv in the loop then vectorization is possible.

Theorem 3.2. *Given a loop with a valid specification, such that every* send *occurs before the matching* recv *in the loop body, and V that divides N.*

1. *All executions in $[\![LP]\!]^{Vec(V)}$ are data race free.*

2. *All executions in $[\![LP]\!]^{Vec(V)}$ and $[\![LP]\!]^{Seq}$ are functionally equivalent.*

Proof. Because every send occurs before the matching recv, every execution that may occur in $[\![LP]\!]^{Vec(V)}$ can also occur in $[\![LP]\!]^{Spec}$. That is, we can construct a specification order sequence in which the computational steps occur in the same order and in which the happens-before relation on the vectorized sequence are more restrictive than those in the specification order sequence. Hence all vectorized sequences are data race free because all specification order sequences are data race free (Theorem 3.1). Moreover, every vectorized execution is functionally equivalent to a specification order sequence and thus functionally equivalent to $[\![LP]\!]^{Seq}$ (Theorem 3.1). □

3.5. Implementation

This section explains how our verification approach is implemented as part of the VerCors toolset. As discussed in Section 2.3 our toolset encodes the input programs into Viper programs [JKM+14] that then are verified by the Silicon verifier [JKM+14, Vip17, HKMS13, MSS16]. This section briefly describes how this encoding works for the loops specified with an iteration contract.

The main idea is to encode the loop specifications into method contracts. So every loop LP annotated with an iteration contract is encoded by a call to the method loop_main, whose contract encodes the application of the Hoare logic rule for parallel loops, instantiated for the specific iteration contract.

```
/*@  requires  (\forall*  int j; 0<=j && j<N; pre(j));
     ensures   (\forall*  int j; 0<=j && j<N; post(j)); @*/
  loop_main(int N, free(LP)));
```

where \forall* is the tool notation for the universal separating conjunction $\bigstar_{j=0}^{N-1}$ and the method free returns the list of all non-local variables in the loop LP.

We also need to verify that every iteration respects its iteration contract. This is encoded by a method, parametrized by the loop variable, containing the loop body, and specified by the iteration contract.

```
/*@  requires  (0<=j && j<N) ** pre(j);
     ensures   post(j); @*/
  loop_body(int j, int N, free(LP))){ body(j); }
```

where $body(j)$ is the body of the loop LP and j is its iteration variable.

Within the body there may be send and recv statements in the following general form:

```
//@ L_s: if (g_s(j)) { send φ(j) to L_r, d; }
//@ L_r: if (g_r(j)) { recv ψ(j) from L_s, d; }
```

We keep the guards untouched, but the send and recv statements are replaced by method calls as follows:

```
//@ L_s: if (g_s(j)) { send_s_to_r (j, N, free(φ(j)); }
//@ L_r: if (g_r(j)) { recv_s_to_r (j, N, free(ψ(j)); }
```

where

```
//@ requires φ(j);                    //@ ensures ψ(j);
send_s_to_r (int j, int N, free(LP)));  recv_s_to_r (int j, int N, free(LP)));
```

Finally, we need to check that the proof obligations in Equation 3.2 and 3.3 hold.

3.6. Related Work

Verification of High-level Parallel Constructs

Recently, almost all major programming languages have been extended to support high-level parallelization constructs. Verification of these high-level

constructs has been investigated in different ways. Salamanca et al. [SMA14] present a runtime checker for loop-carried data dependences in OpenMP programs. Compared to their approach, we propose a static approach to detect loop-carried dependencies. Thus issues can already be detected at compile-time.

Radoi et al. [RD13] employ the restricted thread structure of parallel loops to specialize a set of static data race detection techniques and make them practical for verification of Java 8 loop-parallelism mechanism [Sta17]. In comparison, their method cannot distinguish between vectorized and parallel loop executions, while our approach proposes different specification patterns for each of these executions. Also, they use a specialized data race techniques for Java 8 collections, while we investigate the problem in a more general context.

Barthe et al. [BCG+13] propose a new program synthesis technique which produces SIMD code for a given innermost loop. They exploit the *relational verification* approach to prove the functional equivalence of the generated SIMD code and the original sequential code, while we employ permission-based separation logic to prove such an equivalence for both vectorized and parallel loop executions.

Automated Loop Verification

Gedell et al. [GH06] employ automated first-order reasoning in order to deal with parallelizable loops instead of interactive proof techniques, such as induction. They transform a loop into a universally quantified update of state changes by the loop body. The extraction of quantified state update for a particular loop iteration is intuitively similar to the idea of iteration contracts in our method. Their technique only works for parallelizable loops where there is no loop-carried dependence, while our iteration contracts idea addresses both dependent and independent loops.

Parallelizing Compilers

From the parallelizing compilers perspective, our approach can complement the current static dependence analysis techniques. Specifically, in case of *input-dependent semantics* where static analysis cannot decide whether a loop is independent or not [OR12, RPR07].

3.7. Conclusion and Future Work

This chapter presents how to verify compiler directives about loop parallelization. Each loop is specified by its iteration contract and in the presence of loop-carried dependences, additional send/recv annotations are added to the iteration specifications to capture the loop-carried data dependences. We prove that loops without send/recv annotations are parallelizable, and for a specific pattern of send/recv annotations the loop is vectorizable. Using iteration contracts, we are able to prove that if a specified loop is parallelizable, vectorizable, or it is inherently sequential.

The described method is modular in the sense that it allows us to treat any parallel loop as a statement, thus nested loops can be dealt with simply by giving them their own iteration contract. In addition to the verification of compiler directives, our approach can be employed to detect possible loop parallelizations even where (in)dependence cannot be determined from static analysis of program text (e.g. for loops with non-affine accesses).

As future work one can investigate how the verifier and the parallelizing compiler can support each other. We believe this support can work in both ways. First of all, the parallelizing compiler can use verified annotations to know about dependencies without analyzing the code itself. Conversely, if the compiler performs an analysis then it could emit its findings as a specification template for the code, from which a complete specification can be constructed. This might extend to a set of techniques for automatic generation of iteration contracts.

CHAPTER 4

PARALLEL PROGRAMMING LANGUAGE (PPL)

"The limits of my language means the limits of my world."

– Ludwig Wittgenstein

Parallel programming is notorious for its error-proneness and difficulty [Lee06]. This becomes especially important when parallel programs are used in critical systems. To parallelize a program, developers must deal with a lot of low-level details to effectively utilize operating system services (e.g. thread creation, termination, synchronization, and scheduling) and hardware resources (e.g. processing and memory resources). In fact all these low-level details might differ from one operating system and hardware platform to another. One approach to alleviate the task of parallelization is to simplify it by hiding low-level complexities from developers, so they only need to focus on crafting the high-level parallelization model. This approach is commonly used in developing high-performance financial and scientific applications [JFY99, Ope17a, Ope17e, RB04, BAMM05] and it is often called deterministic parallel programming.

One way to write deterministic parallel programs is to augment a sequential program with parallelization annotations (typically in the form of compiler directives) that indicate which part of the code can be parallelized. In Chapter 3, we particularly discussed the application of this approach for loop parallelization. However, the approach is not restricted to loops. In fact, current annotations libraries (e.g. OpenMP [ope17d] and OpenACC [ope17b]) are sufficiently flexible to express parallelization over almost any partitioning of a sequential program.

The high-level parallel program defined over the sequential program is later consumed by a parallelizing compiler that generates a low-level parallel program to be executed on a particular platform. A platform change often can be easily handled only by re-compiling the high-level parallel program for the new platform.

Despite the popularity of deterministic parallel programming, the semantics

The content of this chapter is based on the following publications of the author: "A Verification Technique for Deterministic Parallel Programs" [DBH17a] and its extended version [DBH17b].

of parallelization annotations is still informally described in natural language. This results in the incorrect use of annotations and syntactic redundancies, and also hinders the development of static analysis techniques. This chapter formalizes the syntax and semantics of a core Parallel Programming Language (PPL) that captures the main features of deterministic parallel programming. This can be used as a basis for the development of static analysis techniques for deterministic parallel programs; for instance in the next chapter we build up our verification technique for deterministic parallel programming on top of this formalization. In addition, the chapter discusses how OpenMP programs, as a commonly used form of deterministic parallel programming, are translated into PPL programs.

The main contributions of this chapter are the following:

- the syntax and an operational semantics of a core language, PPL, which captures the main forms of parallelization constructs in deterministic parallel programs; and
- an encoding algorithm for the translation of OpenMP programs into PPL programs.

Outline. This chapter is organized as follows. We first give the required background on the deterministic parallel programming by discussing some OpenMP examples. Then Section 4.2 explains the syntax and semantics of PPL. Section 4.3 presents how an OpenMP program is encoded into a PPL program. Finally we end the chapter by discussing related work in Section 4.4 and conclusion and future work in Section 4.5.

4.1. Introduction to OpenMP

This section illustrates the most important OpenMP features by discussing four examples.

OpenMP programs are sequential C or Fortran programs which are augmented by OpenMP compiler directives (*pragmas*). In this thesis we focus on the OpenMP programs that are written in C; however the encoding algorithm and our verification technique can be extended to Fortran as well.

The pivotal parallelization annotation in OpenMP is *omp parallel* which determines the parallelizable code block (called *parallel region*). Threads are

Listing 4 Sequential composition of parallel loops

```
1  void ShiftedAddition( int N,int a[ ],int b[ ],int c[ ],int d[ ] )
2  {
3    #pragma omp parallel
4    {
5      #pragma omp for
6      for( int i  = 0; i  < N; i++ ) //Loop L1
7        { c[  i  ]  = a[ i  ];  }
8      #pragma omp for
9      for( int i  = 0; i  < N; i++ ) //Loop L2
10       { d[  i  ]  = c[ i +1 ] + b[ i  ];  }
11   }
12 }
```

forked upon entering a parallel region and joined back into a single thread at the end of the region[1]. There is a default number of threads for each parallel region. This is modifiable by the developer via the OpenMP API, omp_set_num_thread. The code block inside a parallel region is parameterized by thread identifiers and is shared among all the forked threads. Additional OpenMP annotations are required, to precisely determine how this workload is shared between the threads. We explain these additional annotations by discussing them on four example OpenMP programs. Although we explain the most popular parallelization annotations in OpenMP, we do not cover all aspects of OpenMP. To learn more about OpenMP, we refer to [ope17d].

Sequential Composition of Parallel Loops. The example in Listing 4 shows a parallel region that contains two for-loops L1, L2. The loops are marked as *omp for*. This means that the programmer believes that there is no loop-carried data dependence between the iterations of the loop, thus it is safe to execute them in parallel. Relying on this annotation, the OpenMP compiler distributes the loop iterations among the threads of the parallel region.

By default, OpenMP assumes that there exists an implicit barrier at the end of each parallel loop. Therefore no thread executing the loop L1 can

[1] In practice threads might fork once and then be synchronized by inserting barriers at the end of parallel regions. This avoids frequent forking and joining of threads and yields more efficient execution of the program.

Listing 5 Fusion of parallel loops

```
1  void add(int N, int a[ ], int b[ ], int c[ ], int d[ ]){
2    #pragma omp parallel
3    {
4      #pragma omp for schedule(static) nowait
5      for( int i = 0; i < N; i++ ) //Loop L1
6        { c[ i ] = a[ i ]; }
7      #pragma omp for schedule(static)
8      for( int i = 0; i < N; i++ ) //Loop L2
9        { d[ i ] = c[ i ] + b[ i ]; }
10   }
11 }
```

proceed to the loop L2, unless all threads executing L1 have already finished their executions and reached the implicit barrier. This guarantees that two parallel loops are executing sequentially, which is essential when there is a data dependence between the two parallel loops. Without the implicit barrier, the parallel execution of the iterations from the first and the second loop creates a data race situation.

In OpenMP, all variables which are not local to a parallel region are considered as *shared* by default, unless they are explicitly declared as private (using *private* clause) when they are passed to a parallel region.

Fusion of Parallel Loops. The example in Listing 5 shows a slightly different version of the program discussed in Listing 4. With this example, we explain how OpenMP annotations can be extended with additional *clauses*. These clauses help OpenMP users to have more precise control over the behaviour of threads in the parallel region.

Compared to the previous example, the important change is that here the data dependence on the array c is between the identical iterations of the loop L1 and L2. This enables a more efficient execution, in which instead of sequentializing the loops L1 and L2, only the identical iterations of the loops are executed sequentially. This means that the implicit barrier at the end of the loop L1 can be removed if we can guarantee that the same iterations in both loops are executed by the same threads in the parallel region. This corresponds to the

Listing 6 Vectorized loops in OpenMP

```
1  void add(int N,int M, int a[ ],int b[ ],int c[ ])
2  {
3    #pragma omp parallel
4    {
5      #pragma omp simd simdlen(M)
6      for( int i = 0; i < N; i++ ) //Loop L1
7        { c[ i ] = a[ i ] * b[ i ]; }
8      #pragma omp for simd simdlen(M)
9      for( int i = 0; i < N; i++ ) //Loop L2
10       { c[ i ] = a[ i ] * b[ i ]; }
11   }
12 }
```

parallel execution of the loop which is the result of the *fusion* of the loop L1 and L2.

In OpenMP, this delicate behaviour is defined by the combined use of the nowait and schedule(static) clauses. More specifically, the nowait clause removes the implicit barrier at the end of the loop L1 and the schedule(static) clause guarantees the static thread scheduling.

Parallel Composition of Parallel Loops. Listing 7 presents how the parallel execution of two parallel regions is defined in OpenMP. The example consists of three parallel regions: P_1 in lines 4-13, P_2 in lines 15-23, and P_3 in lines 25-27. Similar to the previous examples, the behaviour of each thread is defined by further OpenMP compiler directives. Here we use omp sections which defines the blocks of the code (marked by omp section) which are executed in parallel. For example, two threads are forked upon entering the parallel region P_1, one executes the method add and the other one executes the method mul. Note that the bodies of the methods are parallel regions too. So the threads executing add and mul methods, fork more threads upon entering the parallel region P_2 and P_3. The parallel region P_2 is our fusion example and the parallel region P_3 is a single parallel loop where omp parallel for is a shorthand for having an omp parallel with a single omp for.

The examples in Listing 4, 5 and 7, demonstrate three different ways to compose the parallel and sequential blocks in OpenMP. The next section presents how these three forms of composition are captured by three block composition operators in our parallel programming language.

Vectorization. Since OpenMP 4.0, the support for the *Single Instruction Multiple Data (SIMD)* execution model has been added to the OpenMP standard. In this execution model, one instruction operates on a vector of data elements instead of a single data element. This is a well-known technique to speed up vector arithmetic, specifically in scientific applications. The number of data elements that one vector instruction can handle, i.e. the *vectorization size*, varies from one processor architecture to another.

Listing 6 shows two examples of loop vectorization in OpenMP. The first example uses the `omp simd` annotation to vectorize the loop `L1`. Based on this annotation, the iterations of the loop are partitioned into smaller chunks. The size of each chunk is equal to the vectorization size given by the extra clause `simdlen` (i.e. `M` in this example). Loop execution is defined as the sequential execution of chunks where each chunk is executed in a vectorized fashion. The second loop shows the other form of OpenMP vectorization using the `omp for simd` annotation. In this case, the loop execution is defined in a similar way; however, this time the iteration chunks are executed in parallel instead of sequentially. In both forms, whenever `M` does not divide `N`, the loop is padded with sufficient dummy iterations such that it becomes divisible by `M`.

4.2. Syntax and Semantics of PPL

This section discusses our parallel programming language, PPL, which is presented as a core language for deterministic parallel programming. In essence, PPL is a language for composition of code blocks. We identify three kinds of *basic blocks*: a *parallel block*, a *vectorized block* and a *sequential block*. Basic blocks are composed by three binary block composition operators: *sequential composition*, *parallel composition* and *fusion composition* where the fusion composition allows two parallel basic blocks to be merged into one.

Listing 7 Parallel composition of parallel regions

```
1  #define N 100
2  void main(){
3  int a[ N ], b[ N ], c[ N ], d[ N ], e[ N ];
4  #pragma omp parallel
5  {
6    #pragma omp sections
7    {
8      #pragma omp section
9        add(N, a, b, c, d);
10     #pragma omp section
11       mul(N, a, b, e);
12   }
13 }}
14 void add(int L, int a[ ], int b[ ], int c[ ], int d[ ]){
15 #pragma omp parallel
16 {
17   #pragma omp for schedule(static) nowait
18   for(int i = 0; i < L; i++ ) //Loop L1
19     { c [ i ] = a [ i ]; }
20   #pragma omp for schedule(static)
21   for(int i = 0; i < L; i++) //Loop L2
22     { d [ i ] = c [ i ] + b [ i ]; }
23 }}
24 void mul(int L, int a[ ], int b[ ], int c[ ]){
25     #pragma omp parallel for
26     for(int i=0; i<L; i++) //Loop L3
27       { c [ i ] = a [ i ] * b [ i ]; }
28 }
```

4.2.1 Syntax

Figure 4.1 presents the PPL syntax. The basic building block of a PPL program is a *block*. Each block has a single entry point and a single exit point. Blocks are composed using three binary composition operators: *parallel composition* ||,

Parallel Programming Language:

Block	::=	$(\text{Block} \mid\mid \text{Block}) \mid (\text{Block} \oplus \text{Block}) \mid (\text{Block} \fatsemi \text{Block}) \mid \text{Par(N) S} \mid \text{S}$
S	::=	s;S | skip
s	::=	ass | if (b) {S} else {S} | while (b) {S} | Vec(N) V
V	::=	$b \Rightarrow$ ass;V | skip
ass	::=	$v := e \mid v := \text{mem}(e) \mid \text{mem}(e) := v$
b	::=	$b \wedge b \mid \neg b \mid e = e \mid e \leq e \mid \ldots$
e	::=	$v \mid gv \mid n \mid e + e \mid e - e \mid \ldots$
v	::=	thread local variables
gv	::=	program global variables
n	::=	integer constants

Figure 4.1: Abstract syntax for Parallel Programming Language

fusion composition $\oplus$ and *sequential composition* $\fatsemi$. The entry block of the program is the outermost block. *Basic blocks* are: a *parallel* block Par (N) S; a *vectorized* block Vec (N) V; and a *sequential* block S, where N denotes the number of parallel threads, i.e., the block's *parallelization level*, S is a sequence of statements and V is a sequence of guarded assignments $b \Rightarrow$ ass. N is a global variable of type integer assumed to be positive, finite and constant within the program.

We assume a restricted syntax for fusion composition such that the operands of the composition are parallel basic blocks with the same parallelization levels. Each basic block has a local read-only variable tid $\in [0..\text{N})$ called *thread identifier* where N is the block's parallelization level. We generalize the term *iteration* to refer to the computations of a single thread in a basic block. So a parallel or vectorized block with parallelization level N has N iterations.

For simplicity, threads have access to a single shared array, which we refer to as *shared memory*, sh. To enable threads to share different regions in the shared memory, we require the existence of a number of shared global variables which are accessible to all threads. This we refer to as *program store*, γ. In this way threads can effectively share which locations in the shared memory they use. For example a user-defined shared array in the shared memory is modeled by its base address, which is a global variable in the program store, and an offset which is typically a thread local variable. For simplicity, we assume that the program store remains constant throughout the program. The local variables accessible to a single thread are modeled as a thread local store which we refer

to as *private memory*, σ.

A thread may update its local variables by performing a local computation ($v := e$), or by reading from the shared memory ($v := \mathsf{mem}(e)$). A thread may also update the shared memory by writing one of its local variables to it ($\mathsf{mem}(e) := v$).

4.2.2 Semantics

The behaviour of PPL programs is described using a small-step operational semantics. Throughout, we assume the existence of the finite domains: VarName, the set of variable names, Loc, the set of memory locations, Val, the set of all values including the memory locations, and $[0..N)$ for thread identifiers. We write $+\!\!+$ to concatenate two statement sequences ($S +\!\!+ S$). To define the program state, we use the following definitions.

$sh \in \mathsf{SharedMem} \triangleq \mathsf{Loc} \to \mathsf{Val}$	shared memory
$\gamma \in \mathsf{Store} \triangleq \mathsf{VarName} \to \mathsf{Val}$	program store
$\sigma \in \mathsf{PrivateMem} \triangleq \mathsf{VarName} \to \mathsf{Val}$	private memory

Now we define BlockState. We distinguish between various kinds of block states: an initial state Init, composite block states ParC and SeqC, a state in which a parallel basic block should be executed Par, a local state Local in which a vectorized or a sequential basic block should be executed, and a terminated block state Done.

$\mathsf{BlockState} \ni \mathsf{EB} \triangleq$

$\mathsf{Init}(\mathsf{Block})\mid$	initial block states
$\mathsf{ParC}(\mathsf{EB},\mathsf{EB})\mid \mathsf{SeqC}(\mathsf{EB},\mathsf{Block})\mid$	composite block states
$\mathsf{Par}(\mathbb{LS})\mid$	parallel basic block states
$\mathsf{Local}(\mathsf{LS})\mid$	thread local states
Done	terminated block state

The Init state consists of a block statement Block. The ParC state consists of two block states, and the SeqC state contains a block state and a block statement Block; they capture all the states that a parallel composition and a sequential composition of two blocks might be in, respectively. The basic block state Par captures all the states that a parallel basic block Par (N) S might be in during its execution. It contains a mapping $\mathbb{LS} \in [0..N) \to \mathsf{LocalState}$, that maps each thread to its local state, which models the parallel execution of the threads. There are three kinds of local states: a vectorized state Vec, a sequential state

Seq, and a terminated sequential state Done.

$$\mathsf{LocalState} \ni \mathsf{LS} \triangleq \begin{array}{ll} \mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{V}, \sigma, \mathsf{S})| & \text{vectorized basic block states} \\ \mathsf{Seq}(\sigma, \mathsf{S})| & \text{sequential basic block states} \\ \mathsf{Done} & \text{terminated sequential basic block states} \end{array}$$

The Vec block state captures all states that a vectorized basic block Vec (N) V might be in during its execution. It consists of $\Sigma \in [0..\mathsf{N}) \rightarrow \mathsf{PrivateMem}$, which maps each thread to its private memory, the body to be executed V, a private memory σ, and a statement S. As vectorized blocks may appear inside a sequential block, keeping σ and S allows continuation of the sequential basic block after termination of the vectorized block. To model vectorized execution, the state contains an auxiliary set $\mathsf{E} \subseteq [0..\mathsf{N})$ that models which threads have already executed the current instruction. Only when E equals $[0..\mathsf{N})$, the next instruction is ready for execution. Finally, the Seq block state consists of private memory σ and a statement S.

We model the *program state* as a triple of block state, program store and shared memory $(\mathsf{EB}, \gamma, sh)$ and *thread state* as a pair of local state and shared memory (LS, sh). The program store is constant within the program and it contains all the global variables (e.g. the initial address of arrays). To simplify our notation, each thread receives a copy of the program store as part of its private memory when it initializes (the rules **Init Par** and **Init Seq**). The operational semantics is defined as a transition relation between program states: $\rightarrow_p \subseteq (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{SharedMem}) \times (\mathsf{BlockState} \times \mathsf{Store} \times \mathsf{SharedMem})$, (Figure 4.2), using an auxiliary transition relation between thread local states $\rightarrow_s \subseteq (\mathsf{LocalState} \times \mathsf{SharedMem}) \times (\mathsf{LocalState} \times \mathsf{SharedMem})$, (Figure 4.3), and a standard transition relation $\rightarrow_{ass} \subseteq (\mathsf{PrivateMem} \times \mathsf{S} \times \mathsf{SharedMem}) \times (\mathsf{PrivateMem} \times \mathsf{SharedMem})$ to evaluate assignments, (Figure 4.4). The semantics of expression e and boolean expressions b over private memory σ, written $\mathcal{E}[\![e]\!]_\sigma$ and $\mathcal{B}[\![b]\!]_\sigma$ respectively, is standard and not discussed any further. We use the standard notation for function update: given a function $f : A \rightarrow B$, $a \in A$, and $b \in B$:

$$f[a := b] = x \mapsto \begin{cases} b & , \ x = a \\ f(x), & \text{otherwise} \end{cases}$$

Program execution starts in a program state $(\mathsf{Init}(\mathsf{Block}), \gamma, h)$ where Block is the program's entry block. Depending on the form of Block, a transition is made into an appropriate block state, leaving the shared memory unchanged.

$$\frac{}{\mathsf{Init}(\mathsf{Block}_1||\mathsf{Block}_2),\gamma,sh \rightarrow_p \mathsf{ParC}(\mathsf{Init}(\mathsf{Block}_1),\mathsf{Init}(\mathsf{Block}_2)),\gamma,sh}\ [\textbf{Init ParC}]$$

$$\frac{}{\mathsf{Init}(\mathsf{Block}_1;\mathsf{Block}_2),\gamma,sh \rightarrow_p \mathsf{SeqC}(\mathsf{Init}(\mathsf{Block}_1),\mathsf{Block}_2),\gamma,sh}\ [\textbf{Init SeqC}]$$

$$\frac{}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 \oplus \mathsf{Par(N)}\ \mathsf{S}_2),\gamma,sh \rightarrow_p \mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 +\!\!+ \mathsf{S}_2),\gamma,sh}\ [\textbf{Init Fuse}]$$

$$\frac{\mathsf{LS},sh \rightarrow_s \mathsf{LS}',sh'}{\mathsf{Local}(\mathsf{LS}),\gamma,sh \rightarrow_p \mathsf{Local}(\mathsf{LS}'),\gamma,sh'}\ [\textbf{Lift Seq}]$$

$$\frac{\mathbb{LS} \triangleq \lambda t \in [0..\mathsf{N}).\mathsf{Seq}(\gamma[\mathsf{tid} := t],\mathsf{S})}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}),\gamma,sh \rightarrow_p \mathsf{Par}(\mathbb{LS}),\gamma,sh}\ [\textbf{Init Par}]$$

$$\frac{}{\mathsf{Init}(\mathsf{S}),\gamma,sh \rightarrow_p \mathsf{Local}(\mathsf{Seq}(\gamma[\mathsf{tid} := 0],\mathsf{S})),\gamma,sh}\ [\textbf{Init Seq}]$$

$$\frac{\mathsf{EB}_1,\gamma,sh \rightarrow_p \mathsf{EB}_1',\gamma,sh'}{\mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2),\gamma,sh \rightarrow_p \mathsf{ParC}(\mathsf{EB}_1',\mathsf{EB}_2),\gamma,sh'}\ \textbf{[ParC Step1]}$$

$$\frac{\mathsf{EB}_2,\gamma,sh \rightarrow_p \mathsf{EB}_2',\gamma,sh'}{\mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2),\gamma,sh \rightarrow_p \mathsf{ParC}(\mathsf{EB}_1,\mathsf{EB}_2'),\gamma,sh'}\ \textbf{[ParC Step2]}$$

$$\frac{}{\mathsf{ParC}(\mathsf{Done},\mathsf{Done}),\gamma,sh \rightarrow_p \mathsf{Done},\gamma,sh}\ \textbf{[ParC Done]} \qquad \frac{}{\mathsf{Local}(\mathsf{Done}),\gamma,sh \rightarrow_p \mathsf{Done},\gamma,sh}\ [\textbf{Local Done}]$$

$$\frac{\mathsf{EB},\gamma,sh \rightarrow_p \mathsf{EB}',\gamma,sh'}{\mathsf{SeqC}(\mathsf{EB},\mathsf{Block}),\gamma,sh \rightarrow_p \mathsf{SeqC}(\mathsf{EB}',\mathsf{Block}),\gamma,sh'}\ \textbf{[SeqC Step]}$$

$$\frac{}{\mathsf{SeqC}(\mathsf{Done},\mathsf{Block}),\gamma,sh \rightarrow_p \mathsf{Init}(\mathsf{Block}),\gamma,sh}\ [\textbf{SeqC Done}]$$

$$\frac{i \in \mathsf{dom}(\mathbb{LS}) \quad \mathbb{LS}(i),sh \rightarrow_s \mathsf{LS}',sh'}{\mathsf{Par}(\mathbb{LS}),\gamma,sh \rightarrow_p \mathsf{Par}(\mathbb{LS}[i := \mathsf{LS}']),\gamma,sh'}\ [\textbf{Par Step}] \qquad \frac{\forall i \in \mathsf{dom}(\mathbb{LS}).(\mathbb{LS}(i) = \mathsf{Done})}{\mathsf{Par}(\mathbb{LS}),\gamma,sh \rightarrow_p \mathsf{Done},\gamma,sh}\ [\textbf{Par Done}]$$

Figure 4.2: Operational semantics for program execution

The evaluation of a ParC state non-deterministically evaluates one of its block states (i.e. EB_1 or EB_2), the evaluation of a sequential block is done by evaluating the local state. The evaluation of a SeqC state evaluates its block state EB step by step when this evaluation is done, the subsequent block is initiated.

The evaluation of a parallel basic block is defined by the rules **Par Step** and **Par Done**. To allow all possible interleavings of the threads in the block's thread pool, each thread has its own local state LS, which can be executed independently, modeled by the mapping $\mathbb{LS}$. A thread in the parallel block terminates if there is no more statement to be executed and a parallel block terminates if all threads executing the block are terminated.

$$\frac{}{\mathsf{Seq}(\sigma, \mathsf{while}\ (b)\ \{\mathsf{S}\}; \mathsf{S}'), sh \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{if}\ (b)\ \{\mathsf{S} +\!\!+ \mathsf{while}\ (\mathsf{b})\ \{\mathsf{S}\}\}\ \mathsf{else}\ \{\mathsf{skip}\}; \mathsf{S}'), sh}\ \textbf{[While]}$$

$$\frac{\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\}; \mathsf{S}), sh \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}_1 +\!\!+ \mathsf{S}), sh}\ \textbf{[if}^{\mathbf{true}}\textbf{]}$$

$$\frac{\neg\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\}; \mathsf{S}), sh \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}_2 +\!\!+ \mathsf{S}), sh}\ \textbf{[if}^{\mathbf{false}}\textbf{]}$$

$$\frac{\sigma, \mathsf{ass}, sh \rightarrow_{ass} \sigma', sh'}{\mathsf{Seq}(\sigma, \mathsf{ass}; \mathsf{S}), sh \rightarrow_s \mathsf{Seq}(\sigma', \mathsf{S}), sh'}\ [\textbf{Ass}] \qquad \frac{}{\mathsf{Seq}(\sigma, \mathsf{skip}), sh \rightarrow_s \mathsf{Done}, sh}\ \textbf{[Seq Done]}$$

$$\frac{\Sigma \triangleq \lambda t \in [0..\mathsf{N}).\sigma[\mathsf{tid} := t]}{\mathsf{Seq}(\sigma, \mathsf{Vec(N)\ V}; \mathsf{S}), sh \rightarrow_s \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}), sh}\ [\textbf{Init Vec}]$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \mathcal{B}[\![b]\!]_{\Sigma(i)} \quad \Sigma(i), \mathsf{ass}, sh \rightarrow_{ass} \sigma', sh'}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}), sh \rightarrow_s \mathsf{Vec}(\Sigma[i := \sigma'], \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}), sh'}\ \textbf{[Vec Step1]}$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \neg\mathcal{B}[\![b]\!]_{\Sigma(i)}}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}), sh \rightarrow_s \mathsf{Vec}(\Sigma, \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}), sh}\ \textbf{[Vec Step2]}$$

$$\frac{}{\mathsf{Vec}(\Sigma, \mathsf{dom}(\Sigma), b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}), sh \rightarrow_s \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}), sh}\ \textbf{[Vec Sync]}$$

$$\frac{}{\mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{skip}, \sigma, \mathsf{S}), sh \rightarrow_s \mathsf{Seq}(\sigma, \mathsf{S}), sh}\ \textbf{[Vec Done]}$$

Figure 4.3: Operational semantics for thread execution

The evaluation of sequential basic block's statements as defined in Figure 4.3 is standard except when it contains a vectorized basic block. A sequential basic block terminates if there is no instruction left to be executed (**Seq Done**). The execution of a vectorized block (defined by the rules **Init Vec**, **Vec Step**, **Vec Sync** and **Vec Done** in Figure 4.3) is done in lock-step, i.e. all threads execute the same instruction and no thread can proceed to the next instruction until all are done, meaning that they all share the same program counter. As explained, we capture this by maintaining an auxiliary set, E, which contains the identifier of the threads that have already executed the vector instruction (i.e. the guarded assignment $b \Rightarrow \mathsf{ass}$). When a thread executes a vector instruction, its thread identifier is added to E (rules **Vec Step**). The semantics of vector instructions (i.e. guarded assignments) is the semantics of assignments if the guard evaluates to true and it does nothing otherwise. When all threads have executed the current vector instruction, the condition $\mathsf{E} = \mathsf{dom}(\Sigma)$ holds, and execution moves on to the next vector instruction of the block (with an empty auxiliary set) (rule **Vec**

$$\frac{}{\sigma, v := e, sh \rightarrow_{ass} \sigma[v := \mathcal{E}[\![e]\!]_\sigma], sh}\ [\textbf{LAss}]$$

$$\frac{}{\sigma, v := \mathsf{mem}(e), sh \rightarrow_{ass} \sigma[v := sh(\mathcal{E}[\![e]\!]_\sigma)], sh}\ [\textbf{rdsh}]$$

$$\frac{}{\sigma, \mathsf{mem}(e) := v, sh \rightarrow_{ass} \sigma, sh[\mathcal{E}[\![e]\!]_\sigma := v]}\ [\textbf{wrsh}]$$

Figure 4.4: Operational semantics for assignments

Sync).The semantics of assignments as defined in Figure 4.4 is standard and does not require further discussion.

4.3. OpenMP to PPL Encoding

This section discusses the encoding of OpenMP programs into PPL. Later in the next chapter, we use this encoding to demonstrate how OpenMP programs as an example of deterministic parallel programs are encoded into PPL and then verified. Before discussing the encoding algorithm, we define a core grammar which captures a commonly used subset of OpenMP [AF11], then we present an encoding algorithm for that subset.

Figure 4.5 presents the OMP grammar which supports the OpenMP annotations: omp parallel, omp for, omp simd, omp for simd, omp sections, and omp single. An OMP program is a finite and non-empty list of Jobs enclosed by omp parallel. The body of omp for, omp simd, and omp for simd, is a for-loop. The body of omp single is either an OMP program or it is a sequential code block SpecS. The omp sections block is a finite list of omp section sub-blocks where the body of each

OMP	::=	#pragma omp parallel $[clause]^*$ {Job$^+$}	|
Job	::=	#pragma omp for $[clause]^*$ {for-loop {SpecS}}	|
		#pragma omp simd $[clause]^*$ {for-loop {SpecS}}	|
		#pragma omp for simd $[clause]^*$ {for-loop {SpecS}}	|
		#pragma omp sections $[clause]^*$ {Section$^+$}	|
		#pragma omp single {SpecS |OMP}	
Section	::=	#pragma omp section {SpecS |OMP}	|
SpecS	::=	a list of sequential statements with a contract	
$clause$	::=	allowed OpenMP clause	

Figure 4.5: OpenMP Core Grammar

```
1  Def For as for(i..N*M){SpecS(i)}
2  Def Par as Par(N∗M){SpecS(tid)}
3  Def WhileVec as
4  while(i ∈ [0..N))
5    Vec(tid ∈ [0..M)) SpecS(i∗M+tid)
6  Def ParVec as
7  Par(tid1 ∈ [0..N))
8    Vec(tid2 ∈ [0..M))
9      SpecS(tid1∗M+tid2)
10
11 encode p = compose (translate p)
12 translate  xs = map m xs
13 m(omp for clause*,For)
14   =(Par,omp for clause*)
15 m(omp simd simdlen(M) clause*,For)
16   =(WhileVec,omp simd clause*)
17 m(omp for simd simdlen(M) clause*,For)
18   =(ParVec,omp for simd clause*)
19 m(omp sections clause*, xs)
20   = (fold ||  (map sec xs),clause*)
21 m(omp single clause*,x)
22   = (map sec x,clause*)
23 sec(omp parallel clause*, Job+)
24   = encode Job+
25 sec(SpecS) = Par(1){SpecS}
26
27 par_able (P1,A1) (P2,A2) = nowait(A1)
28 fusiable (P1,A1) (P2,A2) =
29   omp_for(A1) ∧  omp_for(A2) ∧
30   sched_static (A1) ∧ sched_static(A2) ∧
       nowait(A1)
31
32 bundle _ _ [x] = [x]
33 bundle op cond x:y:ys
34   = x : r  ,  if !(cond x y)
35   = (op x (head r)) : (tail  r),  else
36   where r = bundle op cond (y:ys)
37 compose ys =
38    let  p1 = bundle ⊕ fusiable ys in
39    let  p2 = bundle ||  par_able  p1 in
40    fold  ⨟ p2
```

Figure 4.6: Encoding of a commonly used subset of OpenMP programs into PPL programs

omp section is either an OMP program or it is sequential code block SpecS.

Any OpenMP program which conforms to this grammar can be encoded into a PPL program by the algorithm presented in Figure 4.6. The encoding is divided into a recursive *translate* step and a *compose* step. The translation step recursively encodes all OMP Jobs into their equivalent PPL code blocks without caring about how they will be composed. Later, the compose step conjoins the translated code blocks together to build a PPL program. The translation step is a map, which applies the function m to the list of input tuples and returns a list of output tuples. Depending on the type of the OpenMP construct that is translated, different mapping functions are applicable (line 13-25). Specifically the translation of an OpenMP parallel loop, i.e. omp parallel for clause*, to a parallel basic block in PPL is given in the lines 13-14. Note that for readability purposes, we define For as a short form for a for-loop in OpenMP and Par as

the short form for a parallel basic block in PPL. All the short forms used in the algorithm are defined in lines 1-9. The mapping functions for OpenMP sections, i.e. omp sections clause*, (lines 19-20) and OpenMP single, i.e. omp single clause*, (lines 21-22) call another mapping function sec which is defined in lines 23-25.

The input tuples are in the form (A, C) where A is an OpenMP annotation and C is a code block written in C. The tuple represents an annotated code block in OMP programs. The output tuples are in the form $(P, [A])$ where P is a PPL program and $[A]$ is a list of OpenMP annotations.

The compose step takes as its input a list of tuples in the form $(P, [A])$ (the output of the translate step); then it inserts appropriate PPL composition operators between adjacent tuples in the list provided certain conditions hold. To properly bind tuples to the composition operators, the operators are inserted in three individual passes; one pass for each composition operator, based on the binding precedence of the operators from high to low as follows: $; < || < \oplus$.

Operator insertion is done by the function bundle (lines 32-36). In each pass bundle consumes the input list recursively. Each recursive call takes the two first tuples of the list and inserts a composition operator if the tuples satisfy the conditions of the composition operator; otherwise, it moves one tuple forward and starts the same process again.

For each composition operator the conditions are different. The conditions for parallel and fusion compositions are checked by the functions fusiable and par_able. The fusion composition is inserted between two consecutive tuples $(P_i, [A_i])$ and $(P_j, [A_j])$ where both $[A_i]$ and $[A_j]$ are omp for annotations, the clauses of both annotations include schedule(static), and the clauses of $[A_i]$ include nowait. The parallel composition is inserted between any two tuples in the program where the clauses of the first tuple include nowait. Otherwise, the sequential composition is inserted. The final outcome is a single merged tuple $(P, [A])$ where P is the result of the encoding and $[A]$ can be eliminated.

For example, the OpenMP program in Listing 7 is encoded by the presented encoding algorithm into the following PPL program:

$$\mathcal{P}\left\{\begin{array}{c} \left(\underbrace{\mathsf{Par(L)\quad c[i]{=}a[i];}}_{\mathsf{B_1}} \quad \oplus \quad \underbrace{\mathsf{Par(L)\quad c[i]{=}c[i]{+}b[i];}}_{\mathsf{B_2}}\right) \\ || \\ \overbrace{\mathsf{Par(L)\quad d[i]{=}a[i]{*}b[i];}}^{\mathsf{B_3}} \end{array}\right.$$

Program $\mathcal{P}$ contains three parallel basic blocks B_1, B_2 and B_3. The fusion of B_1 and B_2 creates a composite block that is enclosed by the parentheses. Then the composite block is composed with the basic block B_3 using the parallel composition operator.

4.4. Related Work

There is a long history of work on the formalization of parallelism and concurrency. Compared to the abstractions such as Communicating Sequential Processes (CSP) [Hoa78], Calculus of Communicating Systems (CCS) [Mil82] and Algebra of Communicating Processes (ACP) [BK84] and other similar formalizations, our work is less abstract. However, we believe the abstraction level of PPL serves its purposes. On one hand it is sufficiently abstract to capture the main forms of deterministic parallel programs and on the other hand it is sufficiently concrete that it allows the real-world deterministic parallel programming libraries such as OpenMP to be directly translated into PPL.

The need for the formalization of deterministic parallelization has also been addressed by Lu and Scott in [LS11]. Their goal is to present a unified formal definition for different types of determinism in the context of parallel programming. For this purpose, they use a *history-based-semantics* that is conceptually similar to the trace semantics that we developed to formalize the semantics of loop parallelizations in Chapter 3. Compared to them we present a syntax and an operational semantics for a core deterministic parallel programming language while they do not present a specific programming language. Our PPL language addresses practical parallel programming constructs such as vectorized blocks and fusion of parallel blocks that enable our language to capture the behaviour of real-world parallel programming constructs. This has been demonstrated by presenting an encoding algorithm and an implementation for translating OpenMP programs into PPL. From this perspective their work remains rather high-level and no direct encoding or mapping to the real-world parallel programming languages has been presented.

4.5. Conclusion and Future Work

We presented PPL, a core language for deterministic parallel programming. PPL is defined as a language for the composition of code blocks. We distinguish between three kinds of basic blocks: a parallel block, a vectorized block and a sequential block. The basic blocks in PPL can be composed by three binary block composition operators: sequential composition, parallel composition and fusion composition to construct larger code blocks. A small-step operational semantics was presented for PPL. Moreover, we illustrated how a commonly used subset of OpenMP are encoded into PPL.

In the next chapter we discuss how data race freedom and functional correctness are verified for PPL programs. We also explain how OpenMP programs can be verified by first being translated into PPL using the encoding algorithm presented in this chapter.

As future direction, the encoding algorithm can be extended to support other popular deterministic parallel programming languages and libraries such as Cilk [cli17]. The language itself may also be extended to support a wider range of OpenMP programs. OpenMP *task parallelism* and *atomic operations* are among those features that can be added to the language.

CHAPTER 5

VERIFICATION OF DETERMINISTIC PARALLEL PROGRAMS

"When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

– Fred Brooks

This chapter presents a verification technique to prove the data race freedom and the functional correctness of deterministic parallel programs. We develop our verification technique over the core language for deterministic parallel programming (PPL) that was formalized in the previous chapter. Moreover, we show the practical usability of our approach, by presenting how the technique can be applied to the verification of a commonly used subset of OpenMP programs. Particularly we discuss how the OpenMP example in Listing 7 is specified, encoded into PPL and then verified using our verification technique.

To be able to verify a PPL program, each basic block in the program has to be specified by an *iteration contract* [BDH]. Additionally, the program itself should also be specified by a global contract. Previously in Chapter 3 we used iteration contracts to reason about the correctness of loop parallelizations. In this chapter we use iteration contracts to specify the basic blocks in a PPL program.

To verify a specified PPL program, we show that the block compositions are data race free. This is done by proving that for all *independent iterations* (i.e. the iterations that might run in parallel) all accesses to shared memory are non-conflicting, meaning that they are either disjoint or they are read accesses. Then we show that given the data race freedom of all block compositions, the PPL program can be linearized at the level of its basic blocks. This results in a variant of the PPL program in which all the basic blocks are composed sequentially. This reduces our verification problem to prove the data race freedom and functional correctness of the linearized variant of the PPL program. The linearized variant can be verified by showing that each basic block is data race free and functionally correct with respect to its iteration contract and proving the overall functional correctness of the PPL program using standard separation logic rules for sequential programs.

The content of this chapter is based on the following publication of the author: "A Verification Technique for Deterministic Parallel Programs" [DBH17a].

The main contributions of this chapter are the following:

- a verification approach for reasoning about the data race freedom and functional correctness of PPL programs;
- a soundness proof that all verified PPL programs are indeed data race free and functionally correct with respect to their contracts; and
- tool supports that addresses the complete process from the encoding of the OpenMP program into PPL to the verification of the generated PPL programs.

Outline. This chapter is organized as follows. We first present our verification approach in Section 5.1. The soundness of our technique is discussed in Section 5.2. Then in Section 5.3 we explain how our approach is applied to the verification of OpenMP programs. Finally, we conclude with related work, future directions and conclusion.

5.1. Verification Method

To verify PPL programs, we assume each basic block is specified by an iteration contract. We distinguish between two kinds of formulas in an iteration contract: resource formulas (in permission-based separation logic) and functional formulas (in first-order logic). For an individual basic block, if its iteration contract is proven correct, then the basic block is data race free and it is functionally correct with respect to its iteration contract.

To verify the correctness of the program, using standard permission-based separation logic rules, the contracts of all composite blocks should be given. However, our verification approach requires only the basic blocks to be specified at the cost of an extra proof obligation that ensures that the shared memory accesses of all iterations which are not ordered sequentially are non-conflicting (i.e. they are disjoint or they are read accesses). If this condition holds, the correctness of the PPL program can be derived from the correctness of a linearized variant of the program. The rest of this section discusses the formalization of our approach.

To verify a program, we require each basic block to be specified by an iteration contract. An iteration contract consists of: a resource contract $rc(i)$, and a functional contract $fc(i)$, where i is the thread identifier of the block (i.e.

the iteration variable). The functional contract consists of a precondition $P(i)$, and a postcondition $Q(i)$. We also require the program to be globally specified by a contract G which consists of the resource contract of the program $RC_{\mathcal{P}}$ and the functional contract of the program $FC_{\mathcal{P}}$ with the precondition $P_{\mathcal{P}}$ and the postcondition $Q_{\mathcal{P}}$.

5.1.1 Verification of Basic Blocks

This section discusses how we verify a single basic block. In the next section we discuss how the PPL programs that are constructed by the composition of basic blocks are verified.

There are three types of basic blocks: a sequential block, a vectorized block and a parallel block. Verification of sequential blocks using Hoare logic rules is standard and we do not explain it any further. The parallel blocks are verified by the rule **ParBlock** presented in Figure 5.1. This is an adaptation of the rule **ParLoop** (See Section 3.3). We use the same rule for the verification of vectorized blocks in case there is no inter-iteration data dependences. Verification of vectorized blocks in the presence of inter-iteration data-dependencies requires an extension to the technique that enables inter-iteration permission transfer only among the iterations inside the vectorized block and not with other blocks. This would be quite similar to the permission transfer annotations developed for the specification and verification of vectorized loops in Chapter 3.

To verify basic blocks we extend the standard separation logic with the rule **ParBlock**. In the rule, $rc(i)$ is the resource contract and $P(i)$ and $Q(i)$ are iteration's pre- and postcondition of the basic block Par (N) S respectively. N is the number of block's iterations. S is the body of the parallel basic block and S(i) is the body of the i^{th} iteration of the parallel basic block. The rule states that a parallel basic block is correct with respect to its iteration contract if each iteration respects its contract and the universal separating conjunction over the contracts of all iterations is satisfiable. Note that the presence of data race in the basic block makes the universal separating conjunction unsatisfiable as the sum over the permission fractions on the shared location exceeds one.

5.1.2 Verification of Composite Blocks

After discussing the verification of basic blocks, this section presents our approach to reason about PPL programs.

$$\frac{\forall i \in [0..\mathsf{N}).\{rc(i) \star P(i)\}\mathsf{S}(i)\{rc(i) \star Q(i)\}}{\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc(i) \star P(i)\}\mathsf{Par(N)}\ \mathsf{S}\{\bigstar_{i=0}^{\mathsf{N}-1}\ rc(i) \star Q(i)\}}\ [\textbf{ParBlock}]$$

Figure 5.1: Proof rule for the verification of basic blocks

Let $\mathbb{P}$ be the set of all PPL programs and $\mathcal{P} \in \mathbb{P}$ be an arbitrary PPL program, assuming that each basic block in $\mathcal{P}$ is identified by a unique label. We define $\mathbb{B}_{\mathcal{P}} = \{b_1, b_2, ..., b_n\}$, as the finite set of the basic block labels of the program $\mathcal{P}$. For a basic block b with the parallelization level m, we define a finite set of iteration labels $I_b = \{0^b, 1^b, ..., (m-1)^b\}$ where i^b indicates the i^{th} iteration of the block b. Let $\mathbb{I}_{\mathcal{P}} = \bigcup_{b \in \mathbb{B}_{\mathcal{P}}} I_b$ be the finite set of all iterations of the program $\mathcal{P}$.

To state our proof rule, we first define the set of all iterations which are not ordered sequentially, the *incomparable iteration pairs*, $\mathfrak{I}_{\perp}^{\mathcal{P}}$ as:

$$\mathfrak{I}_{\perp}^{\mathcal{P}} = \{(i^{b_1}, j^{b_2}) | i^{b_1}, j^{b_2} \in \mathbb{I}_{\mathcal{P}} \wedge b_1 \neq b_2 \wedge i^{b_1} \not\prec_e j^{b_2} \wedge j^{b_2} \not\prec_e i^{b_1}\}$$

where $\prec_e \subseteq \mathbb{I}_{\mathcal{P}} \times \mathbb{I}_{\mathcal{P}}$ is the least partial order which defines an extended happens-before relation. The extension addresses the iterations which are happens-before each other because their blocks are fused. We define $\prec_e$ based on two partial orders over the basic blocks of the program: $\prec \subseteq \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$ and $\prec_{\oplus} \subseteq \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$. The former is the standard happens-before relation of blocks where they are sequentially composed by $;$ and the latter is a happens-before relation with respect to the fusion composition $\oplus$. They are defined by means of an auxiliary partial order generator function $\mathcal{G}(\mathcal{P}, \delta) : \mathbb{P} \times \{;, \oplus\} \rightarrow \mathbb{B}_{\mathcal{P}} \times \mathbb{B}_{\mathcal{P}}$ such that: $\prec = \mathcal{G}(\mathcal{P}, ;)$ and $\prec_{\oplus} = \mathcal{G}(\mathcal{P}, \oplus)$. We define $\mathcal{G}$ as follows:

$$\mathcal{G}(\mathcal{P}, \delta) = \begin{cases} \mathbb{G} \cup \{(b', b'') | b' \in \mathbb{B}_{\mathcal{P}'} \wedge b'' \in \mathbb{B}_{\mathcal{P}''}\}, & \text{if } \mathcal{P} = \mathcal{P}' \bullet \mathcal{P}'' \ \wedge\ \delta = \bullet \\ \mathbb{G}, & \text{if } \mathcal{P} = \mathcal{P}' \bullet \mathcal{P}'' \ \wedge\ \delta \neq \bullet \\ \emptyset, & \text{if } \mathcal{P} \in \{\mathsf{Par(N)\ S, S}\} \end{cases}$$

where $\mathbb{G} = \mathcal{G}(\mathcal{P}', \delta) \cup \mathcal{G}(\mathcal{P}'', \delta)$.

The function $\mathcal{G}$ computes the set of all iteration pairs of the input program $\mathcal{P}$ which are in relation with respect to the given composition operator δ. This computation is basically a syntactical analysis over the input program. Now we define the extended partial order $\prec_e$ as:

$$\forall i^b, j^{b'} \in \mathbb{I}_{\mathcal{P}}. i^b \prec_e j^{b'} \quad \Leftrightarrow \quad (b \prec b') \quad \vee \quad \big((b \prec_{\oplus} b') \ \wedge\ (i = j)\big)$$

$$\frac{\begin{array}{c}\forall(i^b, j^{b'}) \in \mathfrak{I}^{\mathcal{P}}_{\perp}.(RC_{\mathcal{P}} \Rightarrow rc_b(i) \star rc_{b'}(j)) \\ \{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} blin(\mathcal{P}) \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}\end{array}}{\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} \mathcal{P} \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}} \; [\mathbf{b\text{–}linearize}]$$

Figure 5.2: Proof rule for the b-linearization of PPL programs.

This means that the iteration i^b happens-before the iteration $j^{b'}$ if b happens-before b' (i.e. b is sequentially composed with b') or if b is fused with b' and i and j are corresponding iterations in b and b'.

We extend the program logic with the proof rule **b-linearize**. We first define the *block level linearization* (*b-linearization* for short) $blin : \mathbb{P} \to \mathbb{P}_{;}$ as a program transformation which substitutes all non-sequential compositions by a sequential composition. We define $\mathbb{P}_{;}$ as a subset of $\mathbb{P}$ in which only the sequential composition $;$ is allowed as a composition operator.

Figure 5.2 presents the rule **b-linearize**. In the rule, $rc_b(i)$ and $rc_{b'}(j)$ are the resource contracts of two different basic blocks b and b' where $i^b \in I_b$ and $j^{b'} \in I_{b'}$. Application of the rule results in two new proof obligations. The first ensures that all shared memory accesses of all incomparable iteration pairs (the iterations that may run in parallel) are non-conflicting (i.e. all block compositions in $\mathcal{P}$ are memory safe). This reduces the correctness proof of $\mathcal{P}$ to the correctness proof of its b-linearized variant $blin(\mathcal{P})$.

The correctness of the program $blin(\mathcal{P})$ is proved by the rule **sequentialize** in Figure 5.3 where Block_b denotes any basic block (of type parallel, vectorized, or sequential) with label $b \in \mathbb{B}_{\mathcal{P}}$. The use of the rule requires to prove that (1) each basic block in $\mathcal{P}$ is correct against its iteration contract for which the standard separation logic rules extended with the rule **ParBlock** can be used, and (2) the *sequential variant* of $\mathcal{P}$ is correct with respect to the contract of $\mathcal{P}$ (i.e. $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\} seq(\mathcal{P}) \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$). Sequential variant of $\mathcal{P}$, $seq(\mathcal{P})$ is a version of $blin(\mathcal{P})$ in which all parallel and vectorized basic blocks are executed sequentially. A sequential execution of a parallel or a vectorized block is defined as an execution in which all statements of the iteration i execute before all statements of the iteration $i + 1$. The correctness of $seq(\mathcal{P})$ is proved by the standard separation logic rules for sequential programs that are not discussed any further. Note that we reduce the correctness proof of $blin(\mathcal{P})$ to the correctness proof of its sequential variant. Intuitively this is correct because the functional behaviour of any proven correct basic block is equivalent to the behaviour of its sequential execution. The next section formally proves this

$$\frac{\begin{array}{c}\forall b \in \mathbb{B}_{\mathcal{P}}.\{\bigstar_{i\in[0..N_b)} rc_b(i)\}\mathsf{Block}_b\{\bigstar_{i\in[0..N_b)} rc_b(i)\} \\ \{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}seq(\mathcal{P})\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}\end{array}}{\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}blin(\mathcal{P})\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}}\ [\textbf{sequentialize}]$$

Figure 5.3: Proof rule for sequential reduction of b-linearized PPL programs.

intuitive argument.

5.2. Soundness

Next we show that a PPL program with provably correct iteration contracts and a global contract that is provable in the standard separation logic extended with the rules **ParBlock**, **sequentialize**, and **b-linearize** is indeed data race free and functionally correct with respect to the specifications. To show this, we prove the soundness of the rules **ParBlock**, **sequentialize**, and **b-linearize**, as well as the data race freedom of all verified programs.

For the soundness of the rule **b-linearize**, we show that for each *program execution* there exists a corresponding *b-linearized execution* with the same functional behaviour (i.e. they end in the same terminal state if they start in the same initial state) if all independent iterations are non-conflicting. From the rule's assumption, we know that if the precondition holds for the initial state of the b-linearized execution (which is also the initial state of the program execution) then its terminal state satisfies the postcondition. As both executions end in the same terminal state, the postcondition thus also holds for the program execution. Then we prove that there exists a matching b-linearized execution for each program execution: we first show that any valid program execution can be normalized with respect to the program order, and second that any normalized execution can be mapped to a b-linearized execution.

The soundness of the rule **sequentialize** relies on the soundness of the rule **ParBlock** and the soundness of separation logic rules for sequential programs. The soundness of the rule **ParBlock** is proved based on the rule's assumption that each iteration satisfies its iteration contract, and the fact that the universal separating conjunction over all iterations of the block is satisfiable. The latter implies that all accesses of all iterations of a verified basic block to the shared locations are non-conflicting.

To formalize these arguments, we first define: an *execution*, an *instrumented*

execution, and a *normalized execution*. We assume all program's blocks including basic and composite blocks have a block label and all program's statements have a statement label. There exists a total order over the block labels, which is the order in which they appear in the program.

Definition 5.1. *(Execution). An* execution *of a program $\mathcal{P}$ is a finite sequence of state transitions* $\mathsf{Init}(\mathcal{P}), \gamma, sh \rightarrow_p^* \mathsf{Done}, \gamma, sh'$.

To distinguish between valid and invalid executions, we instrument our operational semantics with *shared memory masks*. A shared memory mask models the access permissions to every shared memory location. It is defined as a map from locations to fractions $\pi : \mathsf{Loc} \rightarrow \mathsf{Frac}$ where Frac is the set of fractions ($[0, 1]$). Any fraction $(0, 1)$ is read and 1 is write permission. The instrumented semantics ensures that each transition has sufficient access permissions to the shared memory locations that it accesses.

To instrument the operational semantics of PPL, we first add a shared memory mask π to all block state constructors (Init, ParC, SeqC and so on) and local state constructors (Vec, Seq and Done). Then we extend the operational semantics rules such that in each block initialization state with shared memory mask π, an extra premise should be discharged, which states that there are $n \geq 2$ shared memory masks $\pi_1, ..., \pi_n$, one for each newly initialized state, such that $\Sigma_i^n \pi_i \leq \pi$; The shared memory masks are carried along by the computation and termination transitions without any extra premises, while in the termination transitions, shared memory masks of the terminated blocks are forgotten as they are not required after termination.

Figure 5.5,5.6 and 5.7 show the instrumented semantics of PPL. We use $\rightarrow_{p,i}$, $\rightarrow_{s,i}$, $\rightarrow_{ass,i}$ to denote program, thread and assignment transitions in the instrumented semantics respectively. If a transition cannot satisfy its premises, it blocks.

Definition 5.2. *(Instrumented Execution). An* instrumented execution *of a program $\mathcal{P}$ is a finite sequence of state transitions* $\mathsf{Init}(\mathcal{P}, \pi), \gamma, sh \rightarrow_{p,i}^* \mathsf{Done}(\pi), \gamma, sh'$ *where the set of all instrumented executions of $\mathcal{P}$ is written as $\mathbb{IE}_{\mathcal{P}}$.*

Lemma 5.1. *Assuming that (1). $\vdash \forall(i^b, j^{b'}) \in \mathfrak{I}_{\perp}^{\mathcal{P}}.RC_{\mathcal{P}} \Rightarrow rc_b(i) \star rc_{b'}(j)$ and (2). $\forall b \in \mathbb{B}_{\mathcal{P}}.\{\bigstar_{i \in [0..N_b)} rc_b(i)\}\mathsf{Block}_b\{\bigstar_{i \in [0..N_b)} rc_b(i)\}$ are valid for a program $\mathcal{P}$ (i.e. every basic block in $\mathcal{P}$ respects its iteration contract), for any execution E of the program $\mathcal{P}$, there exists a corresponding instrumented execution.*

Proof. Given an execution E, we assign shared memory masks to all program states that the execution E might be in. The program's initial state is assigned by a shared memory mask $\pi \leq 1$. Assumption (1) implies that all iterations which might run in parallel are non-conflicting which implies that for all **Init ParC** transitions, there exist π_1 and π_2 such that $\pi_1 + \pi_2 \leq \pi'$ where π' is the shared memory mask of the state in which **Init ParC** evaluates. In all computation transitions, the successor state receives a copy of the shared memory mask of its predecessor. Assumption (2) implies that all iterations of all parallel and vectorized basic blocks are non-conflicting. This implies that for an arbitrary **Init Par** or **Init Vec** transition which initializes a basic block b, there exists $\pi_1, ..., \pi_n$ such that $\Sigma_i^n \pi_i \leq \pi_b$ holds in b's initialization transition, and in all computation transitions of an arbitrary iteration i of the block b the premises of **rdsh** and **wrsh** transitions is satisfiable by π_i. □

Lemma 5.2. *All instrumented executions of a program $\mathcal{P}$ are data race free.*

Proof. The proof proceeds by contradiction. Assume that there exists an instrumented execution that has a data race. Thus, there must be two parallel threads such that one writes to and the other one reads from or writes to a shared memory location e. Because all instrumented executions are non-blocking, the premises of all transitions hold. Therefore, $\pi_1(e) = 1$ holds for the first thread, and $\pi_2(e) > 0$ for the second thread as it either writes or reads. Because the program starts with one single main thread, both threads should have a single common ancestor thread z such that $\pi_x(e) + \pi_y(e) \leq \pi_z(e)$ where x and y are the ancestors of the first and the second thread respectively. A thread only gains permission from its parent; therefore $\pi_1(e) + \pi_2(e) \leq \pi_z(e)$ holds. Permission fractions are in the range $[0, 1]$ by definition, therefore $\pi_1(e) + \pi_2(e) \leq 1$ holds. This implies that if $\pi_1(e) = 1$, then $\pi_2(e) \leq 0$ which is a contradiction.

□

Corollary 5.1. *Assuming that* $\{\bigstar_{i\in[0..N)}\ rc(i)\}\mathsf{Block}\{\bigstar_{i\in[0..N)}\ rc(i)\}$*, for any execution of the basic block* Block *there is an instrumented execution and the basic block* Block *is data race free.*

Proof. The first property follows by instantiating Lemma 5.1 with a program where the PPL program $\mathcal{P}$ is a single basic block Block. The second part is concluded from Lemma 5.2. □

A normalized execution is an instrumented execution that respects the program order, which is defined using an auxiliary labeling function $\mathcal{L} : \mathbb{T} \rightarrow \mathbb{B}_{\mathcal{P}}^{all} \times \mathbb{L}$ where $\mathbb{T}$ is the set of all transitions, $\mathbb{L}$ is the set of labels $\{I, C, T\}$, and $\mathbb{B}_{\mathcal{P}}^{all}$ is the set of block labels (including both composite and basic block labels). We assume all loops are finite; thus the cardinality of $\mathbb{B}_{\mathcal{P}}^{all}$ is finite.

$$\mathcal{L}(t) = \begin{cases} (LB(\mathsf{block}), I), & \text{if } t \text{ initializes a block block} \\ (LB(s), C), & \text{if } t \text{ computes a statement } s \\ (LB(\mathsf{block}), T), & \text{if } t \text{ terminates a block block} \end{cases}$$

where LB accepts both a block or a statement as its input. In the first case, it returns the label of its input block and in the second case it returns the block label to which its input statement belongs. We assume the precedence order $I < C < T$ over $\mathbb{L}$. We say the transition t with label (b, l) is less than t' with label (b', l') if $(b \leq b') \vee (b > b' \rightarrow (l' = T \wedge b \in LB_{sub}(b')))$ where $LB_{sub}(b)$ returns the label set of all blocks of which b is composed. Note that the block label of a composite block is less than the block label of all the blocks from which it composed of. However, the termination transition of the composite block should come after the termination transitions of all the blocks of which it is composed. This is what we describe by the second disjunctive part of the mentioned condition, i.e. $(b > b' \rightarrow (l' = T \wedge b \in LB_{sub}(b')))$.

Definition 5.3. *(Normalized Execution). An instrumented execution labeled by $\mathcal{L}$ is* normalized *if the labels of its transitions are in non-decreasing order.*

We transform an instrumented execution into a normalized one by safely commuting the transitions whose labels do not respect the program order.

Lemma 5.3. *For each instrumented execution of a program $\mathcal{P}$, there exists a normalized execution such that they both end in the same terminal state.*

Proof. Given an instrumented execution $IE = (s_1, t_1) : (s_2, t_2) : IE'$, if $\mathcal{L}(t_1) > \mathcal{L}(t_2)$, a state s_x exists such that a new instrumented execution $IE'' = (s_1, t_2) : (s_x, t_1) : IE'$ can be constructed by swapping two adjacent transitions t_1 and t_2. As the swap is on an instrumented execution which from Lemma 5.2 is data race free, any accesses of t_1 and t_2 to a shared memory location must be reads. Because t_1 and t_2 are adjacent transitions, no other write may happen in between; therefore the swap preserves the functionality of IE, yielding the same terminal state for IE and IE''. Thus, the corresponding normalized execution

of IE, obtained by applying a finite number of such swaps, yields the same terminal state as IE. □

Lemma 5.4. *For each normalized execution of a program $\mathcal{P}$, there exists a b-linearized execution $blin(\mathcal{P})$, such that they both end in the same terminal state.*

Proof. An execution of $blin(\mathcal{P})$ is constructed by applying the map $\mathcal{M}$: BlockState $\rightarrow$ BlockState to each state of a normalized execution. $\mathcal{M}$ is defined as:

$$\mathcal{M}(s) = \begin{cases} \mathsf{Init}(blin(\mathcal{P})), & \text{if } s = \mathsf{Init}(\mathcal{P}) \\ \mathsf{SeqC}(\mathcal{M}(\mathsf{EB}_1), \mathsf{Block}_2), & \text{if } s = \mathsf{ParC}(\mathsf{EB}_1, \mathsf{Init}(\mathsf{Block}_2)) \\ \mathcal{M}(\mathsf{EB}_2), & \text{if } s = \mathsf{ParC}(\mathsf{Done}, \mathsf{EB}_2) \\ \mathsf{SeqC}(\mathsf{Par}(\mathbb{LS}_1), \mathsf{Block}_2), & \text{if } s = \mathsf{Par}(\mathbb{LS}_1 +\!\!+ \mathbb{LS}_2^0) \\ s, & \text{otherwise} \end{cases}$$

where $\mathbb{LS}_2^0$ is the initial mapping of thread local states of Block_2 and $\mathsf{Par}(\mathbb{LS}_1 +\!\!+ \mathbb{LS}_2^0)$ indicates the state of two fused parallel blocks $\mathsf{Par}(\mathbb{LS}_1)$ and $\mathsf{Par}(\mathbb{LS}_2^0)$ where $+\!\!+$ is overloaded and indicates the pairwise concatenation of statements in the local states $\mathbb{LS}_1$ and $\mathbb{LS}_2^0$ (i.e. $\mathsf{S}_1 +\!\!+ \mathsf{S}_2$).

□

Next, we prove that the functional behaviour of any data race free basic block is equivalent to the functional behaviour of its sequential variant. First of all we assume all statements of the basic block are labeled and there is a function LS that returns the label of a particular statement. As we only execute one single basic block, each execution either starts with an **Init Par** transition and ends with **Par Done** or starts with an **Init Seq** following by **Lift Seq** transitions and ends with a **Local Done** transition. Between these fixed transitions all computation transitions occur. Different executions, including the sequential execution, are essentially different orderings of these computation transitions.

To define the sequential execution of a basic block, we use the following labeling function $\mathcal{L}' : \mathbb{T} \rightarrow [0..N) \times \mathbb{SL}$ where $\mathbb{T}$ is the set of all computation transitions, $\mathbb{SL}$ is the set of statement labels and $[0..N)$ is the range of thread identifiers of the basic block Block.

$$\mathcal{L}'(t) = (i, LS(s)), \quad \text{if } t \text{ computes a statement } s$$

we say transition t with label (i, l) is less than t' with label (i', l') if $(i < i') \vee (i = i') \rightarrow (l < l')$.

Lemma 5.5. *For any instrumented execution E of the basic block* Block *(i.e.* $\mathsf{Init}(\mathsf{Block}), \gamma, sh \rightarrow^*_{p,i} \mathsf{Done}, \gamma, sh'$*), there exists a sequential execution (i.e.* $\mathsf{Init}(seq(\mathsf{Block})), \gamma, sh \rightarrow^*_{p,i} \mathsf{Done}, \gamma, sh'$*) such that they both end in the same terminal state.*

Proof. Apart from using a different labeling function $\mathcal{L}'$, the rest of the proof is similar to the proof of Lemma 5.3. □

Definition 5.4. *(Validity of Hoare Triple). The Hoare triple $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}\ \mathcal{P}\ \{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$ is valid if for any execution E (i.e. $\mathsf{Init}(\mathcal{P}), \gamma, sh \rightarrow^*_p \mathsf{Done}, \gamma, sh'$) if $\gamma, sh, \pi \vDash RC_{\mathcal{P}} \star P_{\mathcal{P}}$ is valid in the initial state of E, then $\gamma, sh', \pi \vDash RC_{\mathcal{P}} \star Q_{\mathcal{P}}$ is valid in its terminal state.*

The validity of $\gamma, sh, \pi \vDash RC_{\mathcal{P}} \star P_{\mathcal{P}}$ and $\gamma, sh', \pi \vDash RC_{\mathcal{P}} \star Q_{\mathcal{P}}$ is defined by the semantics of formulas presented in Section 2.2.

Theorem 5.1. *The rule **ParBlock** is sound.*

Proof. For any execution E (i.e. $\mathsf{Init}(\mathsf{Par(N)\ S}), \gamma, sh \rightarrow^*_p \mathsf{Done}, \gamma, sh'$), assume (1) $\gamma, sh, \pi \vDash \bigstar_{i \in [0..N)}\ rc(i) \star P(i)$ holds, and (2) $\forall i \in [0..\mathsf{N}).\ \{rc(i) \star P(i)\}\mathsf{S}(i)\ \{rc(i) \star Q(i)\}$ where $\mathsf{S}(i)$ is the body of the i^{th} iteration of the parallel basic block Par (N) S and i is the block's iteration variable. Suppose that $\gamma, sh', \pi \vDash \bigstar_{i \in [0..N)}\ rc(i) \star Q(i)$ does not hold which means that there exists an iteration i in Par (N) S with the body $\mathsf{S}(i)$ for which $rc(i) \star Q(i)$ does not hold or the universal separating conjunction over all resources is invalid (i.e. $\gamma, sh', \pi \nvDash \bigstar_{i \in [0..N)}\ rc(i)$). The first case contradicts with assumption (2) and the second case contradicts with assumption (1).

□

Theorem 5.2. *The rule **sequentialize** is sound.*

Proof. From assumption (1), Corollary 5.1 and the soundness of the rule **ParBlock**, we conclude that there is an instrumented execution IE_b for each basic block $b \in \mathbb{B}_{\mathcal{P}}$. This implies that there is a corresponding instrumented execution IE for any execution of $blin\ (\mathcal{P})$ as $blin\ (\mathcal{P})$ is the sequential composition of all basic blocks by definition. From Lemma 5.5, there is a sequential execution E_b for each basic block such that IE_b and E_b both end in the same terminal state. Given the only execution E of $seq\ (\mathcal{P})$ (i.e. the sequential execution of $\mathcal{P}$), because both IE and E by definition execute the same basic blocks in the same order if they both start in the same initial state, they both are in the same state after the execution of the same number of basic blocks. Therefore they are in the

same terminal state at the end of their executions. The initial state of IE and E, satisfy the precondition $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}$. From assumption (2) and the soundness of Hoare logic, $\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$ holds in the terminal state of E which thus also holds in the terminal state of IE as they both end in the same terminal state. □

Theorem 5.3. *The rule **b-linearize** is sound.*

Proof. Assume (1). $\vdash \forall(i^b, j^{b'}) \in \mathfrak{I}^{\mathcal{P}}_{\perp}.RC_{\mathcal{P}} \Rightarrow rc_b(i) \star rc_{b'}(j)$ and (2). $\vdash \{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}\ blin(\mathcal{P})\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$. From assumption (2) and the soundness of the rule **sequentialize**, we conclude (3). $\forall b \in \mathbb{B}_{\mathcal{P}}.\{\bigstar_{i\in[0..N_b)} rc_b(i)\}\mathsf{Block}_b\ \{\bigstar_{i\in[0..N_b)} rc_b(i)\}$. Given a program $\mathcal{P}$, implication (3), assumption (1) and, Lemma 5.1 imply that there exists a corresponding instrumented execution IE for any execution of $\mathcal{P}$. Lemma 5.3 and Lemma 5.4 imply that for each instrumented execution IE, there exists a b-linearized execution E' such that both IE and E' end in the same terminal state. The initial states of both IE and E' satisfy the precondition $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}$. From assumption (2) and the soundness of the rule **sequentialize**, $\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$ holds in the terminal state of E' which thus also holds in the terminal state of IE as they both end in the same terminal state. □

Finally, we show that a verified program is indeed data race free.

Proposition 5.1. *A verified program is data race free.*

Proof. Given a program $\mathcal{P}$, with the same reasoning steps mentioned in the Theorem 5.3, we conclude that there exists an instrumented execution IE for $\mathcal{P}$. From Lemma 5.2 all instrumented executions are data race free. Thus, all executions of a verified program are data race free. □

5.3. Verification of OpenMP Programs

Finally, this section discusses the practical applicability of our approach, by showing how it can be used for the verification of OpenMP programs. We demonstrate this in detail on the OpenMP program presented in Section 4.1. More verified OpenMP examples are available online[1]. In Section 4.3, we precisely identified a commonly used subset of OpenMP programs that can be encoded into PPL programs, that also identifies the subset of OpenMP programs that can be verified with our approach.

[1]See the online version of the VerCors toolset at `http://www.utwente.nl/vercors/`.

Program Contract (PC):

```
/*@ invariant  a != NULL && b != NULL && c != NULL && d != NULL && L>0;
invariant  \length(a)==L && \length(b)==L && \length(c)==L && \length(d)==L;
context \forall*  int k; 0 <= k && k < L; Perm(a[k],1/2);
context \forall*  int k; 0 <= k && k < L; Perm(b[k],1/2);
context \forall*  int k; 0 <= k && k < L; Perm(c[k],1);
context \forall*  int k; 0 <= k && k < L; Perm(d[k],1);
ensures \forall  int k; 0 <= k && k < L; c[k]==a[k]+b[k] && d[k]==a[k]*b[k];@*/
```

Iteration Contract 1 (IC_1) of loop L1:

```
/*@ context Perm(c[i],1) ** Perm(a[i],1/4);
ensures c[i]==a[i];  @*/
```

Iteration Contract 2 (IC_2) of loop L2:

```
/*@ context Perm(c[i],1) ** Perm(b[i],1/4);
ensures c[i]==\old(c[i])+b[i];  @*/
```

Iteration Contract 3 (IC_3) of loop L3:

```
/*@ context Perm(d[i],1) ** Perm(a[i],1/4) ** Perm(b[i],1/4);
ensures d[i]==a[i]*b[i];  @*/
```

Figure 5.4: Required contracts for verification of the OpenMP example

We verify OpenMP programs in the following three steps: (1) manually specifying the program (i.e. providing an iteration contract for each loop and writing the program contract for the outermost OpenMP parallel region), (2) automatically encoding of the specified OpenMP program into its PPL counterpart (carrying along the original OpenMP specifications), (3) automatically checking the PPL program against its specifications. Steps two and three have been implemented as part of the VerCors toolset [BH14, BDHO17, Ver17a]. The details of the encoding algorithm are discussed in Section 4.3.

Figure 5.4 shows the required contracts for the parallel composition of the parallel regions presented in Listing 7 of Section 4.1. There are four specifications. The first one is the program contract which is attached to the outermost parallel block. The others are the iteration contracts of the loops L1, L2 and L3. The *requires* and *ensures* keywords indicate pre- and postconditions of each contract, whereas the *context* keyword is a shorthand for both requiring and ensuring the same predicate. We use $**$ and $\backslash forall*$ to denote separating conjunction $\star$ and universal separating conjunction $\bigstar_{i \in I}$ respectively. Before verification, we encode the example into the following PPL program $\mathcal{P}$:

$$
/{*}@\ \mathsf{Program\ Contract}\ @{*}/
$$

$$
\mathcal{P}\begin{cases}
\big(\underbrace{\mathsf{Par(L)\ /*@IC_1@*/\ c[i]=a[i];}}_{\mathsf{B}_1} \oplus \underbrace{\mathsf{Par(L)\ /*@IC_2@*/\ c[i]=c[i]+b[i];}}_{\mathsf{B}_2}\big) \\
\qquad\qquad\qquad\qquad || \\
\qquad\qquad \overbrace{\mathsf{Par(L)\ /*@IC_3@*/\ d[i]=a[i]*b[i];}}^{\mathsf{B}_3}
\end{cases}
$$

Program $\mathcal{P}$ contains three parallel basic blocks B_1, B_2 and B_3 and is verified by discharging two proof obligations: (1) ensure that all shared memory accesses of all incomparable iteration pairs (i.e. all iteration pairs except the identical iterations of B_1 and B_2) are non-conflicting which implies that the fusion of B_1 and B_2 and the parallel composition of $\mathsf{B}_1 \oplus \mathsf{B}_2$ and B_3 are memory safe (2) prove that each parallel basic block by itself satisfies its iteration contract $\forall \mathsf{b} \in \{1, 2, 3\}.\{\bigstar_{i\in[0..L)}\mathsf{IC_b}(i)\}\mathsf{B_b}\ \{\bigstar_{i\in[0..L)}\mathsf{IC_b}(i)\}$, and (3) prove the correctness of the b-linearized variant of $\mathcal{P}$ against its program contract $\{RC_{\mathcal{P}} \star P_{\mathcal{P}}\}\ \mathsf{B}_1 \fatsemi \mathsf{B}_2 \fatsemi \mathsf{B}_3\{RC_{\mathcal{P}} \star Q_{\mathcal{P}}\}$.

We have implemented a slightly more general variant of PPL in the tool that supports variable declarations and method calls. To check the first proof obligation in the tool we quantify over the pairs of blocks that allows the number of iterations in each block to be a parameter rather than a fixed number.

5.4. Related Work

Botincan et al. propose a proof-directed parallelization synthesis, which takes as input a sequential program with a proof in separation logic and outputs a parallelized counterpart by inserting barrier synchronizations [BDJ12, BDJ13]. Hurlin uses a proof-rewriting method to parallelize a sequential program's proof [Hur09a]. Compared to them, we prove the correctness of parallelization by reducing the parallel proof to a b-linearized proof. Moreover, our approach allows verification of sophisticated block compositions, which enables reasoning about state-of-the-art parallel programming languages (e.g. OpenMP) while their work remains rather theoretical.

Raychev et al. use abstract interpretation to make a non-deterministic program (obtained by naive parallelization of a sequential program) deterministic by inserting barriers [RVY13]. This technique over-approximates the possible program behaviours which ends up in a determinization whose behaviour is

implied by a set of rules which decide between feasible schedules rather than the behaviour of the original sequential program. Unlike them, we do not generate any parallel program. Instead we prove that parallelization annotations can safely be applied and that the parallelized program is functionally correct and exhibits the same behaviour as its sequential counterpart.

Barthe et al. synthesize SIMD code given pre- and postconditions for loop kernels in C++ STL or C# BCL [BCG+13]. We alternatively enable verification of SIMD loops, by encoding them into vectorized basic blocks. Moreover, we address the parallel or sequential composition of those loops with other forms of parallelized blocks.

Dodds et al. introduce a higher-order variant of Concurrent Abstract Predicates (CAP) to support modular verification of synchronization constructs for deterministic parallelism [DJP11]. Their proofs use nested region assertions and higher-order protocols, but they do not address the semantic difficulties introduced by these features which make their reasoning unsound.

Salamanca et al. [SMA14] propose a runtime loop-carried dependence checker as an extension to OpenMP which helps programmers to detect hidden data dependencies in *omp parallel for*. Compared to them, we statically detect any violation of data dependencies without any runtime overhead and we address a larger subset of OpenMP constructs.

5.5. Conclusion and Future Work

We have presented a verification technique to reason about the data race freedom and functional correctness of PPL programs. To do so, we extend separation logic with the rules **ParBlock**, **sequentialize** and **b-linearize**. The rule **ParBlock** is used to prove that a single basic block is correct with respect to its iteration contract. The rule **sequentialize** is used to reduce the correctness of b-linearized variant of a PPL program to its sequential counterpart, given the assumption that all basic blocks in the PPL program are correct with respect to their iteration contracts. Finally the rule **b-linearized** proves that the correctness of PPL program can be reduced to the correctness of its b-linearized variant, if all incomparable iterations are non-conflicting. So the correctness of PPL program is reduced in two steps to the correctness proof of its sequential counterpart, which then is provable using the standard separation logic rules for sequential programs.

We illustrated the practical applicability of our technique by discussing how a commonly used subset of OpenMP can be verified using the presented technique.

As future work, we plan to look into adapting annotation generation techniques to automatically generate iteration contracts, including both resource formulas and functional properties. This will lead to fully automatic verification of deterministic parallel programs. Moreover, our technique can be extended to address a larger subset of OpenMP programs by supporting more complex OpenMP patterns for scheduling iterations and *omp task* constructs. We also plan to identify the subset of atomic operations that can be combined with our technique that allows verification of the widely-used reduction operations.

$$\frac{\pi_1 + \pi_2 \leq \pi}{\mathsf{Init}(\mathsf{Block}_1 || \mathsf{Block}_2, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{ParC}(\mathsf{Init}(\mathsf{Block}_1, \pi_1), \mathsf{Init}(\mathsf{Block}_2, \pi_2), \pi), \gamma, sh} \text{ [Init ParC]}$$

$$\frac{}{\mathsf{Init}(\mathsf{Block}_1 ; \mathsf{Block}_2, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{SeqC}(\mathsf{Init}(\mathsf{Block}_1, \pi), \mathsf{Block}_2, \pi), \gamma, sh} \text{ [Init SeqC]}$$

$$\frac{}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 \oplus \mathsf{Par(N)}\ \mathsf{S}_2, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}_1 +\!\!+ \mathsf{S}_2, \pi), \gamma, sh} \text{ [Init Fuse]}$$

$$\frac{\mathsf{LS}, sh \rightarrow_{s,i} \mathsf{LS}', sh'}{\mathsf{Local}(\mathsf{LS}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Local}(\mathsf{LS}', \pi), \gamma, sh'} \text{ [Lift Seq]}$$

$$\frac{\mathbb{LS} \triangleq \lambda t \in [0..\mathsf{N}).\mathsf{Seq}(\gamma[\mathsf{tid} := t], \mathsf{S}, \pi_t) \quad \Sigma_{t=0}^{N} \pi_t \leq \pi}{\mathsf{Init}(\mathsf{Par(N)}\ \mathsf{S}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Par}(\mathbb{LS}, \pi), \gamma, sh} \text{ [Init Par]}$$

$$\frac{}{\mathsf{Init}(\mathsf{S}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Local}(\mathsf{Seq}(\gamma[\mathsf{tid} := 0], \mathsf{S}, \pi), \pi), \gamma, sh} \text{ [Init Seq]}$$

$$\frac{\mathsf{EB}_1, \gamma, sh \rightarrow_{p,i} \mathsf{EB}_1', \gamma, sh'}{\mathsf{ParC}(\mathsf{EB}_1, \mathsf{EB}_2, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{ParC}(\mathsf{EB}_1', \mathsf{EB}_2, \pi), \gamma, sh'} \textbf{ [ParC Step1]}$$

$$\frac{\mathsf{EB}_2, \gamma, sh \rightarrow_{p,i} \mathsf{EB}_2', \gamma, sh'}{\mathsf{ParC}(\mathsf{EB}_1, \mathsf{EB}_2, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{ParC}(\mathsf{EB}_1, \mathsf{EB}_2', \pi), \gamma, sh'} \textbf{ [ParC Step2]}$$

$$\frac{}{\mathsf{ParC}(\mathsf{Done}(\pi_1), \mathsf{Done}(\pi_2), \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Done}(\pi), \gamma, sh} \textbf{ [ParC Done]}$$

$$\frac{}{\mathsf{Local}(\mathsf{Done}(\pi), \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Done}(\pi), \gamma, sh} \text{ [Local Done]}$$

$$\frac{\mathsf{EB}, \gamma, sh \rightarrow_{p,i} \mathsf{EB}', \gamma, sh'}{\mathsf{SeqC}(\mathsf{EB}, \mathsf{Block}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{SeqC}(\mathsf{EB}', \mathsf{Block}, \pi), \gamma, sh'} \textbf{ [SeqC Step]}$$

$$\frac{}{\mathsf{SeqC}(\mathsf{Done}(\pi), \mathsf{Block}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Init}(\mathsf{Block}, \pi), \gamma, sh} \textbf{ [SeqC Done]}$$

$$\frac{i \in \mathsf{dom}(\mathbb{LS}) \quad \mathbb{LS}(i), sh \rightarrow_{s,i} \mathsf{LS}', sh'}{\mathsf{Par}(\mathbb{LS}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Par}(\mathbb{LS}[i := \mathsf{LS}'], \pi), \gamma, sh'} \textbf{ [Par Step]}$$

$$\frac{\forall i \in \mathsf{dom}(\mathbb{LS}).(\mathbb{LS}(i) = \mathsf{Done}(\pi_i))}{\mathsf{Par}(\mathbb{LS}, \pi), \gamma, sh \rightarrow_{p,i} \mathsf{Done}(\pi), \gamma, sh} \textbf{ [Par Done]}$$

Figure 5.5: Instrumented operational semantics for program execution

$$\frac{}{\mathsf{Seq}(\sigma, \mathsf{while}\ (b)\ \{\mathsf{S}\}; \mathsf{S}', \pi), sh \rightarrow_{s,i} \mathsf{Seq}(\sigma, \mathsf{if}\ (b)\ \{\mathsf{S} +\!\!+ \mathsf{while}\ (\mathsf{b})\ \{\mathsf{S}\}\}\ \mathsf{else}\ \{\mathsf{skip}\}; \mathsf{S}', \pi), sh}\ \textbf{[While]}$$

$$\frac{\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\}; \mathsf{S}, \pi), sh \rightarrow_{s,i} \mathsf{Seq}(\sigma, \mathsf{S}_1 +\!\!+ \mathsf{S}, \pi), sh}\ \textbf{[if}^{\mathbf{true}}\textbf{]}$$

$$\frac{\neg\mathcal{B}[\![b]\!]_\sigma}{\mathsf{Seq}(\sigma, \mathsf{if}(b)\{\mathsf{S}_1\}\mathsf{else}\{\mathsf{S}_2\}; \mathsf{S}, \pi), sh \rightarrow_{s,i} \mathsf{Seq}(\sigma, \mathsf{S}_2 +\!\!+ \mathsf{S}, \pi), sh}\ \textbf{[if}^{\mathbf{false}}\textbf{]}$$

$$\frac{\sigma, \mathsf{ass}, sh, \pi \rightarrow_{ass,i} \sigma', sh', \pi}{\mathsf{Seq}(\sigma, \mathsf{ass}; \mathsf{S}, \pi), sh \rightarrow_{s,i} \mathsf{Seq}(\sigma', \mathsf{S}, \pi), sh'}\ [\textbf{Ass}] \quad \frac{}{\mathsf{Seq}(\sigma, \mathsf{skip}, \pi), sh \rightarrow_{s,i} \mathsf{Done}(\pi), sh}\ \textbf{[Seq Done]}$$

$$\frac{\Sigma \triangleq \lambda t \in [0..\mathsf{N}).\sigma[\mathsf{tid} := t] \quad \Pi \triangleq \lambda t \in [0..\mathsf{N}).\pi_t \quad \Sigma_{t=0}^{N}\Pi(t) \leq \pi}{\mathsf{Seq}(\sigma, \mathsf{Vec}(\mathsf{N})\ \mathsf{V}; \mathsf{S}, \pi), sh \rightarrow_{s,i} \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh}\ [\textbf{Init Vec}]$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \mathcal{B}[\![b]\!]_{\Sigma(i)} \quad \Sigma(i), \mathsf{ass}, sh, \Pi(i) \rightarrow_{ass,i} \sigma', sh', \Pi(i)}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh \rightarrow_{s,i} \mathsf{Vec}(\Sigma[i := \sigma'], \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh'}\ \textbf{[Vec Step1]}$$

$$\frac{i \in \mathsf{dom}(\Sigma)\backslash\mathsf{E} \quad \neg\mathcal{B}[\![b]\!]_{\Sigma(i)}}{\mathsf{Vec}(\Sigma, \mathsf{E}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh \rightarrow_{s,i} \mathsf{Vec}(\Sigma, \mathsf{E} \cup \{i\}, b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh}\ \textbf{[Vec Step2]}$$

$$\frac{}{\mathsf{Vec}(\Sigma, \mathsf{dom}(\Sigma), b \Rightarrow \mathsf{ass}; \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh \rightarrow_{s,i} \mathsf{Vec}(\Sigma, \emptyset, \mathsf{V}, \sigma, \mathsf{S}, \Pi, \pi), sh}\ \textbf{[Vec Sync]}$$

$$\frac{}{\mathsf{Vec}(\Sigma, \mathsf{E}, \mathsf{skip}, \sigma, \mathsf{S}, \Pi, \pi), sh \rightarrow_{s,i} \mathsf{Seq}(\sigma, \mathsf{S}, \pi), sh}\ \textbf{[Vec Done]}$$

Figure 5.6: Instrumented operational semantics for thread execution

$$\frac{}{\sigma, v := e, sh, \pi \rightarrow_{ass,i} \sigma[v := \mathcal{E}[\![e]\!]_\sigma], sh, \pi}\ [\textbf{LAss}]$$

$$\frac{l = \mathcal{E}[\![e]\!]_\sigma \quad \pi(l) > 0}{\sigma, v := \mathsf{mem}(e), sh, \pi \rightarrow_{ass,i} \sigma[v := h(\mathcal{E}[\![e]\!]_\sigma)], sh, \pi}\ [\textbf{rdsh}]$$

$$\frac{l = \mathcal{E}[\![e]\!]_\sigma \quad \pi(l) = 1}{\sigma, \mathsf{mem}(e) := v, sh, \pi \rightarrow_{ass,i} \sigma, sh[\mathcal{E}[\![e]\!]_\sigma := v], \pi}\ [\textbf{wrsh}]$$

Figure 5.7: Instrumented operational semantics for assignments

CHAPTER 6

VERIFICATION OF GPGPU PROGRAMS

"The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: divide and conquer."

– Bjarne Stroustrup

In previous chapters, we discussed how high-level parallelization approaches in the form of parallel loops and deterministic parallel programs can be verified. However, execution of those high-level programs requires a translation to lower-level programs that depending on the hardware platform can be multihreaded programs written for example in C, Java, or GPGPU programs. To complete the story, in this chapter we discuss how permission-based separation logic can be used to reason about low-level programs, in particular GPGPU programs.

Graphics processing units (GPUs) originally have been designed to support computer graphics. Their architecture supports fast memory manipulation, and a high processing power by using massive parallelism, making them suitable to efficiently solve typical graphics-related tasks. However, this architecture is also suitable for many other programming tasks, leading to the emergence of the area of *General Purpose GPU (GPGPU) programming*. Initially, this was mainly done in CUDA [JS10], a proprietary GPU programming language from NVIDIA. However, from 2006 onwards, OpenCL [ope17c] has become more and more popular as a new platform-independent, low-level programming language for GPGPU programming. Nowadays, GPUs are used in many different fields such as media processing [CK11], medical imaging [SHT+08], and eye-tracking [Mul12].

GPGPU programming is based on the notion of *kernels*. A kernel consists of a large number (typically hundreds) of parallel threads that all execute the same instructions. The GPU execution model is an extension of the *Single Instruction Multiple Data* (SIMD) execution model[1], in which each thread executes the same instruction but on different data. For efficiency reasons, threads on a GPU device are grouped into *work groups*. Each work group has its own *local memory*,

The content of this chapter is based on the following publications of the author: "Specification and verification of atomic operations in GPGPU programs" [ADBH15].

[1]To be precise, the GPU execution model is Single Instruction Multiple Thread (SIMT), which extends SIMD with more flexibility in the control flow.

shared among all threads in the work group. Further, the kernel has a *global memory*, which is shared among all threads on the GPU device.

Threads within a work group usually synchronize by *barriers*. Atomic operations provide asynchronous updates on shared memory locations (either in global or local memory) and are the only mechanism to support inter-group synchronization in GPU programs. Moreover, atomic operations are also sometimes used for synchronization within a work group, because they enable more flexible parallel behaviours than using barriers alone. For example, the *Parallel add* example in Section 6.3.5 and the *Histogram* example in the Parboil benchmark [SRS+12] benefit from the flexible parallel behaviour of atomic operations.

GPGPU programming enables programmers to use the power of massively parallel accelerator devices to solve computationally intensive problems with a significant speed up. However, the use of massive parallelism also makes the GPU program highly prone to errors such as data races. In GPGPU programming these errors could be very subtle, so that even expert programmers sometimes fail to detect them[2].

Recently, different verification techniques have been developed to reason about data race freedom of GPGPU kernels. We are specifically interested in the technique presented by Blom et al. [BHM14] which besides proving data race freedom, can reason about the functional correctness of GPU kernels. However, the technique lacks the support for atomic operations. In this chapter, we discuss how we extend the technique to reason about kernels that also use atomic operations. The main idea of our work is to adapt the notion of *resource invariants*, as originally introduced for Concurrent Separation Logic (CSL) by O'Hearn, to reason about the behaviour of atomic operations with respect to the GPU memory hierarchy.

Resource invariants capture the properties of shared memory locations. These properties only may be violated by a thread that is in the critical section, and thus has exclusive access to the shared memory locations. Before leaving the critical section, the thread has to ensure that the resource invariants are re-established. Because of the GPU memory hierarchy, shared memory locations can be both in local memory (shared between threads in a single work group) and in global memory (shared between all threads). Therefore, in our approach

[2] An example of these subtle bugs is the data race bug in N-body example shipped with version 2.3 of the CUDA SDK [BCD+15].

we use *group resource invariants* that capture the properties for local shared memory locations, and *kernel resource invariants* to capture the properties for global shared memory locations. For each kernel, there always is a single kernel resource invariant, while for each work group there is a group resource invariant. However, by parameterizing the group resource invariant with the group identifier, this can be specified with a single formula.

Moreover, we discuss how the specifications provided for high-level programs (i.e. iteration contracts) can be translated into low-level kernel specifications. This enables us to verify the semantic equivalence of the high-level program and its low-level translation. It also reduces the annotation overhead as it allows to reuse high-level specifications for low-level code.

To conclude, the main contributions of this chapter are:

- a specification and verification technique that adapts the notion of CSL resource invariants to the GPU memory model and enables us to reason about the data race freedom and the functional correctness of GPGPU kernels containing atomic operations;
- a soundness proof of our approach;
- a demonstration of the usability of our approach by developing automated tool support for it; and
- a translation method to encode iteration contracts into kernel specifications.

Outline. The remainder of this chapter is organized as follows. After some background information on GPGPU programming in Section 6.1, Section 6.2 discusses the syntax and semantics of Kernel Programming Language (KPL). Based on that Section 6.3 explains how that logic and specification language is extended to support atomic operations in GPGPU programs. Section 6.4 formalizes our approach and presents its soundness. Section 6.5 discusses how the approach is implemented in our VerCors toolset. Section 6.6 reveals the connection between iteration contracts and kernel specifications. Finally we conclude this chapter with related work and conclusion in Section 6.7 and 6.8 respectively.

Listing 8 A basic example of a GPU kernel

```
1 __kernel void shift (__global int * a, __global int * b){
2     ltid = get_local_id (0);
3     gsize = get_local_size (0);
4     a[ ltid ] = ltid ;
5     barrier (CLK_GLOBAL_MEM_FENCE);
6     b[ ltid ] = a[ (ltid +1) % gsize ];
7 }
```

6.1. Concepts of GPGPU Programming

A GPU runs hundreds of threads simultaneously. All threads within the same kernel execute the same instruction, but on different data: the Single Instruction Multiple Data (SIMD) execution model. GPU kernels are invoked by a *host* program, typically running on a CPU. Threads are grouped into *work groups* and are identified by thread IDs. There are two different numberings for thread IDs: a global thread ID which determines the identifier of thread with respect to all the running threads and a local thread ID, which does so with respect to a single work group. Both values can be retrieved at any point in the kernel via calling corresponding API functions.

GPUs have three different memory regions: *global*, *local*, and *private* memory. Private memory is local to a single thread, local memory is shared between threads within a work group, and global memory is accessible to all threads in a kernel, and to the host program. Through some configurations in the host program, the data and the kernel source code is moved to the device memory. The host code also determines the number of work groups and the number of threads which are supposed to execute the kernel. The host code launches the kernel and collects the results at the end of its execution.

Threads within a single work group can synchronize by using a *barrier*. This ensures that no thread in the work group can pass the barrier unless all threads reach the barrier. In addition to their synchronization function, barriers in GPU programming also act as memory fences on the local or global memory (or both of them). So passing the barrier ensures that all memory writes on the shared memory have been committed. The memory level to which memory fence function of the barrier applies, can be determined by the programmer via

setting the appropriate barrier flag. The threads within different work groups cannot be synchronized. However, it is possible for all the threads on the device to atomically access to the shared locations in global and local memory. In OpenCL atomic operations are supported by *atomic functions* (See the list of OpenCL atomic functions [Khr]).

Listing 8 shows the code of a kernel that initializes an array b on the global memory in such a way that the position i of the array contains $i + 1$ modulo the length of the array (assumed to be equal to the work group size gsize). To do this, each thread first assigns its thread ID, ltid, to the position i of a temporary array a on the global memory. All threads wait at the barrier and then the position i of array b is assigned by the value of the position $i + 1$ modulo the work group size, gsize of the array a. If the barrier would be removed, there would be a data race on a[i].

Note that although arrays are stored in global memory, the execution of kernel with multiple work groups, when the global thread identifier is used, can result in a data race because the barrier, can not synchronize threads running in different work groups.

6.2. Kernel Programming Language

This section discusses our core kernel programming language that formalizes the main features of GPGPU programming languages (namely OpenCL). The language is an extension of the Kernel Programming Language (KPL) [BCD+12] with support for atomic operations. Over this language, we present our verification technique for reasoning about GPGPU programs.

6.2.1 Syntax

Figure 6.1 presents the syntax for our kernel programming language. For simplicity, in this language, global and local memory are assumed to be single shared arrays. There are two local memory access operations: read from location e_1 in local memory ($v := \mathsf{rdloc}(e_1)$), and write e_2 to location e_1 in local memory $\mathsf{wrloc}(e_1, e_2)$. Similarly, read and write operations in global memory are represented by ($v := \mathsf{rdglob}(e)$) and $\mathsf{wrglob}(e_1, e_2)$, respectively.

With respect to the original KPL language, barriers are different. They are labeled with barrier identifiers bid, which are used to distinguish between different barrier instances. They are also extended with statement bodies to be

Reserved global identifiers (constant within a thread):

$gtid$	Global thread identifier with respect to the kernel
gid	Group identifier with respect to the kernel
$ltid$	Local thread identifier with respect to the work group
$tcount$	The total number of threads in the kernel
$gsize$	The number of threads per work group
ks	The number of groups in the kernel

Kernel language:

$$
\begin{array}{lcl}
b & ::= & \text{boolean expression over global constants and private variables} \\
e & ::= & \text{integer expression over global constants and private variables} \\
\mathsf{S} & ::= & v := e \mid v := \mathsf{rdloc}(e) \mid v := \mathsf{rdglob}(e) \mid \mathsf{wrloc}(e_1, e_2) \mid \mathsf{wrglob}(e_1, e_2) \\
 & \mid & \mathsf{skip} \mid \mathsf{S}_1;\mathsf{S}_2 \mid \mathsf{if}\ (b)\ \{\mathsf{S}_1\}\ \mathsf{else}\ \{\mathsf{S}_2\} \mid \mathsf{while}\ (b)\ \{\mathsf{S}\} \\
 & \mid & \mathsf{atomic}(\mathsf{F})\{\mathsf{S}\} \mid\ bid : \mathsf{barrier}(\mathsf{F})\{\mathsf{S}\} \\
\mathsf{F} & ::= & \mathsf{local} \mid \mathsf{global}
\end{array}
$$

Figure 6.1: Syntax for Kernel Programming Language

executed atomically by all threads as soon as they all reach the barrier. Barriers accept a flag F that denotes which memories are accessed by the body of the barrier. Note that this is different from the barrier flag used in OpenCL where they determine the memories on which the memory fencing function of the barrier applies. However, in our language all memory writes and reads happen immediately; so there is no need to capture that behaviour in our semantics.

Further, we add an atomic block statement to the language, again with a flag F to denote whether it accesses locations in the global or local memory. The OpenCL atomic operations can be easily embedded into this atomic block statement.

6.2.2 Semantics

To describe the behaviour of kernels, we present a small-step operational semantics. Threads on a GPU are executed in *lock-step* fashion. Intuitively, this means that all threads execute the same instruction and no thread can proceed to the next instruction until all are done. So they all share the same program counter. In practice, the model is a bit more complicated. The number of threads, and number and size of work groups are typically larger than the number of actual physical processing units on the device. Therefore, only a

subset of threads in a work group, known as *warp* in NVIDIA architecture [TC13, NVI] or *wavefront* in AMD architecture [AMD, ROA12], execute the same instruction at the same time. The size of this subset (i.e. the number of threads) varies from one vendor to another and from one hardware architecture to another one. This execution model results in the most efficient execution, because in the meantime, data that is used by the next subset of threads can be fetched from or written to memory. However, the specific details of this execution are hardware-specific. As we intend our operational semantics to describe the most general behaviour possible, we therefore consider all possible interleavings between two barriers. The soundness of our verification approach is proven with respect to this most general behaviour, thus any verified property holds for any possible implementation.

Throughout, we assume that we have the sets Gid, T, and Bid of work group, thread and barrier identifiers, with typical members gid, $gtid$, and bid, respectively. Global and local memory are modeled as a single shared array. We assume the existence of the finite domains: VarName, the set of variable names, and Val, the set of all values, which includes the memory locations, Loc, the set of memory locations.

$\sigma \in \mathsf{GlobalMem} \triangleq \mathsf{Loc} \to \mathsf{Val}$	global memory (accessible to all threads)
$\delta \in \mathsf{LocalMem} \triangleq \mathsf{Loc} \to \mathsf{Val}$	local memory (accessible to a work group)
$\gamma \in \mathsf{PrivateMem} \triangleq \mathsf{VarName} \to \mathsf{Val}$	private store (accessible to a single thread)

The state of a kernel KernelState consists of the global memory, and all its group states. The state of each group GroupState consists of local memory, and all its thread states. Finally, the state of a thread ThreadState consists of an instruction, its private state and a tag whether it is running R, or waiting at the barrier $bid \in \mathsf{Bid}$, W_{bid}. Formally, this is defined as follows:

$$
\begin{array}{lcl}
\mathsf{KernelState} & \triangleq & \mathsf{GlobalMem} \times (\mathsf{Gid} \to \mathsf{GroupState}) \\
\mathsf{GroupState} & \triangleq & \mathsf{LocalMem} \times (\mathsf{Lid} \to \mathsf{ThreadState}) \\
\mathsf{ThreadState} & \triangleq & \mathsf{Stmt} \times \mathsf{PrivateMem} \times \mathsf{BarrierTag} \\
\mathsf{BT} \in \mathsf{BarrierTag} & \triangleq & \mathsf{R} \mid \mathsf{W}_{bid}
\end{array}
$$

Below, updates to group and thread states are written using function up-

dates, defined as follows: Given a function $f : A \to B$, $a \in A$, and $b \in B$:

$$f[a := b] = x \mapsto \begin{cases} b & , x = a \\ f(x), & \text{otherwise} \end{cases}$$

The operational semantics of kernel behaviour is defined by the following three relations:

$$\begin{array}{rcl} \to_{\mathcal{K}} & \subseteq & (\mathsf{KernelState})^2 \\ \to_{\mathcal{G},gid} & \subseteq & (\mathsf{GlobalMem} \times \mathsf{GroupState})^2 \\ \to_{\mathcal{T},gtid} & \subseteq & (\mathsf{GlobalMem} \times \mathsf{LocalMem} \times \mathsf{ThreadState})^2 \end{array}$$

Figure 6.2 presents the rules defining these relations. As mentioned above, the operational semantics defines all possible interleavings. Therefore, the kernel state changes if one group changes its state (the rule **Kernel Step**). A group changes its state if one thread changes its state (the rule **Group Step**). A thread can change its state by executing an instruction according to the standard operational semantics rules for imperative languages (the rules **Assign**, **Global/Local Read/Write**). Figure 6.2 only gives the rules for sequential composition (the rules **Sequential Composition 1** and **2**); the rules for conditionals and loops are omitted as they are standard. If a thread enters a barrier (the rule **Barrier Enter**), it enters the "waiting at barrier" state. This is denoted by a thread state with the barrier tag W_{bid} where bid is the barrier identifier on which the thread is blocked. At this state all threads in the group are waiting to *atomically* execute the body of barrier S followed by the rest of kernel statements. Once, at the group level, all threads have entered, the states are simultaneously switched back to running (the rule **Barrier Synchronize**). A thread can also atomically perform all statements in an $\mathsf{atomic}(\mathsf{F})\{\mathsf{S}\}$. The operational semantics of an atomic block is defined by the rule **Atomic Step**. The semantics of expression e over the private store γ in thread $gtid$ is denoted $[\![e]\!]_{\gamma}^{gtid}$; its definition is standard and not discussed further. In the kernel's *initial state*, all memories are empty, and all threads contain the full kernel body as the statement to execute.

6.3. Specification of GPGPU Programs

We first explain our specification method and then we demonstrate it by discussing three example kernels. The first example is a simple kernel with barrier. This explains the method introduced in [BHM14]. Then the second and

third examples discuss how the kernels with atomic operations are specified. In particular, the second example uses a single *atomic add* operation, while the third example illustrates a kernel that uses both a barrier and atomic operations for synchronization.

$$\frac{\Delta(gid) = (\delta, \Gamma) \quad \sigma, \delta, \Gamma \rightarrow_{\mathcal{G},gid} \sigma', \delta', \Gamma'}{\sigma, \Delta \rightarrow_{\mathcal{K}} \sigma', \Delta[gid := (\delta', \Gamma')]} \textbf{[Kernel Step]}$$

$$\frac{\Gamma(ltid) = (\mathsf{S}, \gamma, \mathsf{BT}) \quad \sigma, \delta, (\mathsf{S}, \gamma, \mathsf{BT}) \rightarrow_{\mathcal{T},gid \cdot gsize + ltid} \sigma', \delta', (\mathsf{S}', \gamma', \mathsf{BT}')}{\sigma, \delta, \Gamma \rightarrow_{\mathcal{G},gid} \sigma', \delta', \Gamma[ltid := (\mathsf{S}', \gamma', \mathsf{BT}')]} \textbf{[Group Step]}$$

$$\frac{}{\sigma, \delta, (bid : \mathsf{barrier}(\mathsf{F})\{\mathsf{S}_1\}; \mathsf{S}_2, \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma, \delta, (\mathsf{atomic}(\mathsf{F})\{\mathsf{S}_1\}; \mathsf{S}_2, \gamma, \mathsf{W}_{bid})} \textbf{[Barrier Enter]}$$

$$\frac{\forall ltid \in \mathsf{Lid}.\Gamma(ltid) = (\mathsf{atomic}(\mathsf{F})\{\mathsf{S}_1\}; \mathsf{S}_2, \gamma_{ltid}, \mathsf{W}_{bid})}{\sigma, \delta, \Gamma \rightarrow_{\mathcal{G},gid} \sigma, \delta, \lambda ltid \in \mathsf{Lid}.\Gamma[ltid := (\mathsf{atomic}(\mathsf{F})\{\mathsf{S}_1\}; \mathsf{S}_2, \gamma_{ltid}, \mathsf{R})]} \textbf{[Barrier Synchronize]}$$

$$\frac{\sigma, \delta, (\mathsf{S}, \gamma, \mathsf{R}) \rightarrow^*_{\mathcal{T},gtid} \sigma', \delta', (\epsilon, \gamma', \mathsf{R})}{\sigma, \delta, (\mathsf{atomic}(\mathsf{F})\{\mathsf{S}\}, \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma', \delta', (\epsilon, \gamma', \mathsf{R})} \textbf{[Atomic Step]}$$

$$\frac{}{\sigma, \delta, (v := e, \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma, \delta, (\epsilon, \gamma[v := [\![e]\!]^{gtid}_{\gamma}], \mathsf{R})} \textbf{[Assign]}$$

$$\frac{}{\sigma, \delta, (v := \mathbf{rdglob}(e), \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma, \delta, (\epsilon, \gamma[v := \sigma([\![e]\!]^{gtid}_{\gamma})], \mathsf{R})} \textbf{[Global Read]}$$

$$\frac{}{\sigma, \delta, (v := \mathbf{rdloc}(e), \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma, \delta, (\epsilon, \gamma[v := \delta([\![e]\!]^{gtid}_{\gamma})], \mathsf{R})} \textbf{[Local Read]}$$

$$\frac{}{\sigma, \delta, (\mathbf{wrglob}(e_1, e_2), \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma[[\![e_1]\!]^{gtid}_{\gamma} := [\![e_2]\!]^{gtid}_{\gamma}], \delta, (\epsilon, \gamma, \mathsf{R})} \textbf{[Global Write]}$$

$$\frac{}{\sigma, \delta, (\mathbf{wrloc}(e_1, e_2), \gamma, \mathsf{R}) \rightarrow_{\mathcal{T},gtid} \sigma, \delta[[\![e_1]\!]^{gtid}_{\gamma} := [\![e_2]\!]^{gtid}_{\gamma}], (\epsilon, \gamma, \mathsf{R})} \textbf{[Local Write]}$$

$$\frac{\sigma, \delta, (\mathsf{S}_1, \gamma, \mathsf{BT}) \rightarrow_{\mathcal{T},gtid} \sigma', \delta', (\mathsf{S}'_1, \gamma', \mathsf{BT}')}{\delta, \sigma, (\mathsf{S}_1; \mathsf{S}_2, \gamma, \mathsf{BT}) \rightarrow_{\mathcal{T},gtid} \sigma', \delta', (\mathsf{S}'_1; \mathsf{S}_2, \gamma', \mathsf{BT}')} \textbf{[Sequential Composition 1]}$$

$$\frac{\sigma, \delta, (\mathsf{S}_1, \gamma, \mathsf{BT}) \rightarrow_{\mathcal{T},gtid} \sigma', \delta', (\epsilon, \gamma', \mathsf{BT}')}{\sigma, \delta, (\mathsf{S}_1; \mathsf{S}_2, \gamma, \mathsf{BT}) \rightarrow_{\mathcal{T},gtid} \sigma', \delta', (\mathsf{S}_2, \gamma', \mathsf{BT}')} \textbf{[Sequential Composition 2]}$$

Figure 6.2: Small-step operational semantics rules

6.3.1 Specification Method

To support our verification technique, the user needs to provide the following specifications[3]:

- The *kernel specification* is a triple $(K_{pre}, K_{post}, K_{rinv})$. The K_{pre} and K_{post} specify the kernel pre- and postcondition, respectively. An invocation of a kernel by a host program is correct if the host program holds the necessary resources and fulfills the preconditions. The *kernel resource invariant* K_{rinv} specifies all the resources[4] that are accessed in an atomic operation or a barrier block. The specified resources are accessible for a thread only when it is inside an atomic block.
- The *group specification* is a triple $(G_{pre}, G_{post}, G_{rinv})$, where G_{pre} and G_{post} specify the pre- and postcondition, respectively and G_{rinv} specifies the *group resource invariant* which is parameterized by group identifier. Notice that locations in local memory are only accessible from within the work group and thus the work group always holds write permissions to these locations.
- Permissions and conditions in the work group are distributed over the work group's threads by the *thread specification* (T_{pre}, T_{post}). Because threads within a work group can exchange permissions (at barriers), the resources before and after execution might be different.
- A *barrier specification* (B_{pre}, B_{post}) specifies resources, and a pre- and postcondition for each barrier in the kernel. Besides the functional state right before and after barrier, they also specify how permissions are redistributed over the threads (depending on the barrier flag, these can be permissions on local memory only, on global memory only, or a combination of global and local memory). The barrier precondition B_{pre} specifies the functional property and resources that have to hold when a thread reaches the barrier. The barrier postcondition B_{post} specifies the functional property and resources that are allowed to be assumed after the execution of the barrier to continue the verification of the thread.

Note that the user only has to annotate a kernel resource invariant K_{rinv},

[3]Only group and kernel resource invariants are the contribution of this thesis. However, because they are necessary for understanding the content of this chapter, we discuss the other parts of the specification briefly. For further examples and discussions, we refer to [BHM14].

[4]As introduced earlier in Chapter 2, resources are permission formulas in permission-based separation logic specifying the ownership of threads over the memory locations.

a group resource invariant G_{rinv}, a thread's pre- and postcondition T_{pre} and T_{post} and barrier's pre- and postcondition B_{pre} and B_{post}. We can derive the work groups' pre- and postconditions, i.e. G_{pre} and G_{post}, as the separating conjunction of the pre- and postconditions of all threads belonging to the work group and the work group's resource invariant. Similarly, the kernel's pre- and postcondition, i.e. K_{pre} and K_{post}, can be derived automatically as the separating conjunction of the pre- and postconditions of all work groups belonging to the kernel and the kernel's resource invariant.

6.3.2 Syntax of Formulas in our Specification Language

The basis of our specification language is permission-based separation logic discussed previously in Section 2.2. However, to use that language for specification of GPU kernels, we need to adapt it to support GPU memory model. To do so, instead of one permission predicate to capture thread accesses to the shared memory locations, we use two permission predicates one for the accesses to local memory and the other one for the accesses to global memory. The expressions are also need to be separated based on the reads from the local or global memory.

The syntax of formulas in our specification language is in the following form:

$$
\begin{array}{lll}
E & ::= & \text{expressions (in first-order logic) over global constants,} \\
 & & \text{private variables, } \mathsf{rdloc}(E), \text{ and } \mathsf{rdglob}(E). \\
R & ::= & \mathsf{true} \mid E \mid \mathsf{LPerm}(E,p) \mid \mathsf{GPerm}(E,p) \mid R_1 \star R_2 \mid E \Rightarrow R \mid \bigstar_{v:E(v)} R(v)
\end{array}
$$

$\mathsf{LPerm}(E,p)$ and $\mathsf{GPerm}(E,p)$ capture thread's accesses to local memory and global memory, respectively. The formulas can be conjoined using separating conjunction, guarded by expressions, or quantified over the set of values v for which $E(v)$ is true.

6.3.3 Specification of a Kernel with Barrier

The example in Listing 9 illustrates the specification technique that is used in [BHM14] for reasoning about kernels where barriers are the only synchronization mechanism. The example contains a kernel program annotated with a *thread specification*, plus a *barrier specification* for each barrier. For simplicity, it has a single work group.

The specifications use the keywords ltid to denote the local thread identifier,

Listing 9 An example of a kernel with specifications

```
1  /*@  requires  Perm(a[ltid], write) ** Perm(b[ltid], write);
2       ensures   Perm(a[ltid], write) ** Perm(b[ltid], write);
3       ensures   b[ ltid ] == (ltid+1) % gsize;
4  @*/
5  __kernel void rotate(__global int * a, __global int * b){
6      ltid = get_local_id (0);
7      gsize = get_local_size (0);
8      a[ ltid ] = ltid;
9      barrier (CLK_GLOBAL_MEM_FENCE)
10     /*@  requires  Perm(a[ltid], write) ** Perm(b[ltid], write);
11            requires  a[ ltid ] == ltid;
12            ensures   Perm(a[ltid],1/2) ** Perm(a[(ltid +1) % gsize],1/2)
13                      ** Perm(b[ltid], write);
14            ensures   a[ (ltid +1) % gsize ] == (ltid+1) % gsize;
15     @*/{}
16     b[ ltid ] = a[ (ltid +1) % gsize ];
17 }
```

and gsize to denote the number of threads in each work group. A thread specification specifies the permissions a thread should hold before and after execution, together with the thread's functional behaviour. In the example, write permission to the position ltid of both array a and b is required and it is ensured that the position ltid of array b can be written and contains (ltid+1) % gsize. To illustrate the use of a barrier, the kernel is implemented in such a way that first ltid is assigned to a[ltid] and then access to the array is rotated by synchronization on a barrier, after which the thread reads a[(ltid+1) % gsize]. This rotation is specified with a *barrier specification*, which specifies: (1) how permissions are redistributed over the threads in the work group, and (2) the functional pre- and postconditions that must hold before and after execution of the barrier.

Group specifications capture the resources in global memory that can be used by the threads in a particular work group, including its pre- and postcondition. Notice that the locations defined in local memory are only valid inside the work group and thus the work group always holds write permissions for these

Listing 10 Specification of parallel add in a work group.

```
1  /*@ given int cont[gsize];
2  group invariant  Perm(x,write)**Perm(cont[*],1/2)**x==(\sum cont[*]);
3  requires  Perm(values[ltid ],1/2) **Perm(cont[ltid],1/2)**cont[ltid]==0;
4  ensures   Perm(values[ltid ],1/2) **Perm(cont[ltid],1/2)**
       cont[ltid]==values[ltid];@*/
5  __kernel  void  gpadd(__local  int  * x,  __local  int  * values )
6  {
7     ltid  = get_global_id (0);
8     atomic_add(x,values [ ltid ])  /*@ then { cont[ltid ]=values[ ltid ];  } @*/;
9  }
```

locations. In the kernel specification, resources that are required from the host program along with the necessary preconditions and provided postconditions are specified. An invocation of a kernel by a host program is correct if the host program transfers the necessary resources and fulfills the kernel preconditions.

6.3.4 Specification of a Kernel with Parallel Addition

Listing 10 contains an annotated parallel add kernel, where ltid indicates the local thread identifier. For simplicity, in this example we assume that we have a single work group[5], later we extend our technique also to multiple work groups. We first explain the permission specifications, followed by an explanation of the functional properties (the highlighted annotations).

In Listing 10, each thread atomically adds its contribution (stored in values[ltid]) to the shared variable x. The requires and ensures clauses express a single thread's pre- and postconditions. The precondition specifies that each thread needs to have read permission on its corresponding index of values. Additionally, we specify a group resource invariant for the local shared memory variable x, which expresses that the thread executing the atomic add operation has exclusive write access to x. With this specification, it is straightforward to prove that the program is free of data races, as it is guaranteed that there is only one thread executing the atomic operation and exclusively accessing the shared variable.

[5]The number of work groups is determined in the host code before launching the kernel.

To reason about functional properties, the specification expresses the accumulative contributions of the threads on the shared variable. To track these contributions, we use an array cont[], added as a ghost variable (line 1) to the kernel. A ghost variable (a.k.a. as auxiliary variable) is a specification-only variable, which does not change the control flow of the program and is used only for verification. The idea is that the contribution of each thread (cont[ltid]) is 0 before it executes and is values[ltid] after it finishes, while the invariant $\sum_{i=0}^{\text{gsize-1}} \text{cont}[i] = \text{x}$ is maintained in order to prove that the kernel computes the sum of the values. To make this work, the thread's precondition (line 3) states that each thread obtains a read permission on cont[ltid], in order to be able to use cont in the specifications. Each thread has to track its contribution towards the total amount of x in its own location in the array cont. This is done during the atomic operation by injecting an assignment statement as ghost code (specified as a then clause, line 8). The thread executing atomic_add first adds values[ltid] to x, and then executes the injected ghost code, i.e. cont[ltid]=values[ltid]. To achieve this, the group resource invariant is extended with a half permission on *all* elements of cont, written Perm(cont[*],1/2) where cont[*] is syntactic sugar for the universal quantification of the permissions over all the indices of cont[]. Thus, when the thread ltid at the beginning of the atomic body obtains the resource invariants, it has *twice* a read permission Perm(cont[ltid],1/2), which can be combined into a single write permission Perm(cont[ltid],write).

6.3.5 Parallel Addition with Multiple Work Groups

As a next example, we discuss the specification of a kernel with multiple work groups, which employs both barriers and atomic operations for synchronization. This is a common pattern to avoid bottleneck in the global memory accesses: first all threads in a work group compute an intermediate result in local memory, then the intermediate result is combined with the global result in global memory. It is used, for example, in the parallel implementation of BFS in the Parboil benchmark [SRS+12]. The kernel in Listing 11 is an extension of the previous example, using multiple work groups and a barrier, where ksize denotes the number of work groups. The kernel is implemented by the following steps: (1) each thread atomically adds its element of the global array values to its local accumulator, i.e. a locally shared variable x; (2) all threads within a work group are synchronized by a barrier (line 16); (3) after all threads

Listing 11 Specification of global parallel add.

```
1  /*@ given global int sums[ks]={0}; given local int cont[gsize]={0}, region=0;
2    kernel invariant Perm(r,write)**Perm(sums[*],1/2)**r==(\sum sums[*]);
3    group invariant Perm(region,1/(gsize+1))**Perm(x,region==0?1:1/2)**
4                    Perm(cont[*],1/2)**x==(\sum cont[*]);
5    requires Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
6    requires Perm(cont[ltid],1/2)**cont[ltid]==0;
7    requires ltid==0 ⇒ Perm(sums[gid],1/2)**sums[gid]==0;
8    ensures Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
9    ensures Perm(cont[ltid],1/4)**cont[ltid]==values[gtid];
10   ensures ltid==0 ⇒ Perm(cont[*],1/4)**Perm(sums[gid],1/2);
11   ensures ltid==0 ⇒ sums[gid]==(\sum cont[*]); @*/
12 __kernel void KParallelAdd(__local int * x, __global int * values, __global int * r)
       {
13   gtid = get_global_id(0);
14   ltid = get_local_id(0);
15   atomic_add(x,values[gtid]) /*@ then { cont[ltid]=values[gtid]; } @*/;
16   barrier(CLK_LOCAL_MEM_FENCE)/*@
17     requires Perm(region,1/(gsize+1))**region==0}**Perm(cont[ltid],1/4);
18     ensures Perm(region,1/(gsize+1))**region==1;
19     ensures ltid==0 ⇒ Perm(cont[*],1/4)**x==(\sum cont[*]); @*/
20   { /*@ region=1; @*/ }
21   if (ltid ==0)
22     atomic_add(r,x)/*@ then { sums[gid]=x; } @*/;
23 }
```

have passed the barrier, one thread per work group (here ltid== 0) adds the work group's final value of x to a *globally* shared variable r (line 22). Eventually, r contains the collective contributions of all the threads in the kernel.

Similar to the single work group example, to track the contributions at each step, the kernel program uses ghost arrays cont and sums, with all elements initialized to zero. We use cont to specify the current value of the local variable x . Similarly, the array sums is used to sum up the total accumulated contributions of the work groups. Updating the local cont is explained in the previous example. In a similar way, using the ghost code at line 22, in each work group, the thread with ltid== 0 stores its contribution (the final value of x) to the global sums[gid] (i.e. the index corresponding to the executing work group from

the sums array).

In Listing 11, there are two resource invariants that capture the functional behaviour of the kernel:

1. $\sum_{i=0}^{\mathsf{gsize}-1} \mathsf{cont}[i] = \mathsf{x}$ for each work group; and

2. $\sum_{i=0}^{\mathsf{ksize\text{-}1}} \mathsf{sums}[i] = \mathsf{r}$ for the kernel.

After termination of work group gid, we use the group invariant to conclude that:

$$\mathsf{sums}[\mathsf{gid}] = \sum_{i=\mathsf{gsize}\times\mathsf{gid}}^{\mathsf{gsize}\times\mathsf{gid}+\mathsf{gsize}-1} \mathsf{values}[i] \ .$$

Hence after termination of all work groups we can prove that:

$$\mathsf{r} = \sum_{i=0}^{\mathsf{ksize\text{-}1}} \mathsf{sums}[i] = \sum_{j=0}^{\mathsf{ksize\text{-}1}} \sum_{i=j\times\mathsf{gsize}}^{(j+1)\times\mathsf{gsize}-1} \mathsf{values}[i]$$

Again, we first explain the permission specifications. The permission specifications for values are similar to the specifications in Listing 10. The barrier divides the program into regions, and within a region the distribution of permissions over the threads and the resource invariants does not change. Only when all threads reach the barrier, permissions may be redistributed. This means in particular that a variable that is treated as a shared memory variable in one region, may become unshared in a next region (or vice versa). Thus, resource invariants often depend on the current barrier region. To keep track of the current barrier region, we use a ghost variable region initialized at 0 (line 1). Each thread at all times has read access to this region variable, and whenever all the threads go through the barrier, the region is updated (line 20). The group resource invariant (lines 3-4) specifies that within region 0 (before the barrier), variable x is shared, while in region 1 (after the barrier), x is not shared any more as it is only accessed by the thread with local thread identifier zero, in each work group. The kernel resource invariant (line 2) specifies the write access to the variable r that is a shared variable in global memory. However, only threads with a local thread identifier 0 are able to update r without violating the kernel resource invariant (namely r==(\sum sums[*])). The reason is that only those threads can construct a write permission on sums[gid] to store the contributions. The write permission is constructed by merging two read permissions, one

provided by the kernel resource invariant and the other one by the thread's precondition (in lines 2 and 7 respectively).

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{R[v := e]\}\, v \;:=\; e\, \{R\}}\ \textbf{[Assign]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{LPerm}(e,\pi) \star R[v := L[e]]\}\, v \;:=\; \mathsf{rdloc}(e)\, \{\mathsf{LPerm}(e,\pi) \star R\}}\ \textbf{[LRead]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{LPerm}(e_1,1) \star R[L[e_1] := e_2]\}\, \mathsf{wrloc}(e_1,e_2)\, \{\mathsf{LPerm}(e_1,1) \star R\}}\ \textbf{[LWrite]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{GPerm}(e,\pi) \star R[v := L[e]]\}\, v \;:=\; \mathsf{rdglob}(e)\, \{\mathsf{GPerm}(e,\pi) \star R\}}\ \textbf{[GRead]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{GPerm}(e_1,1) \star R[L[e_1] := e_2]\}\, \mathsf{wrglob}(e_1,e_2)\, \{\mathsf{GPerm}(e_1,1) \star R\}}\ \textbf{[GWrite]}$$

S refers to local memory only.

$$\frac{K_{rinv} \vdash \{P(t) \star G_{rinv}(gid)\}\ \mathsf{S}\ \{G_{rinv}(gid) \star Q(t)\}}{K_{rinv}\,, G_{rinv}(gid) \vdash\ \{P(t)\}\ \mathsf{atomic(local)\{S\}}\ \{Q(t)\}}\ \textbf{[LAtomic]}$$

S refers to global memory only.

$$\frac{G_{rinv}(gid) \vdash \{P(t) \star K_{rinv}\}\ \mathsf{S}\ \{K_{rinv} \star Q(t)\}}{K_{rinv}\,, G_{rinv}(gid) \vdash\ \{P(t)\}\ \mathsf{atomic(global)\{S\}}\ \{Q(t)\}}\ \textbf{[GAtomic]}$$

S, R, and E refer to local memory only.

$$\frac{K_{rinv} \vdash \{\bigstar_{t\in[0..gsize)} R(t) \star G_{rinv}(gid)\}\ \mathsf{S}\ \{G_{rinv}(gid) \star \bigstar_{t\in[0..gsize)} E(t)\}}{K_{rinv}\,, G_{rinv}(gid) \vdash\ \begin{array}{l}\{P(t) \star R(t)\}\\ \mathbf{barrier}(\mathsf{local})\ \ \mathbf{req}\ R(t);\ \ \mathbf{ens}\ E(t);\ \{\mathsf{S}\}\\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[LBarrier]}$$

S, R, and E refer to global memory only.

$$\frac{G_{rinv}(gid) \vdash \{\bigstar_{t\in[0..gsize)} R(t) \star K_{rinv}\}\ \mathsf{S}\ \{K_{rinv} \star \bigstar_{t\in[0..gsize)} E(t)\}}{K_{rinv}\,, G_{rinv}(gid) \vdash\ \begin{array}{l}\{P(t) \star R(t)\}\\ \mathbf{barrier}(\mathsf{global})\ \ \mathbf{req}\ R(t);\ \ \mathbf{ens}\ E(t);\ \{\mathsf{S}\}\\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[GBarrier]}$$

Figure 6.3: Important proof rules

The barrier specification expresses that threads keep read access on region and that the value of region is updated to 1. Moreover, the specification asserts that upon entering the barrier each thread gives up $1/4$ permission to access its contribution element, i.e. cont[ltid]. The barrier redistributes these permissions to the thread with ltid$==0$, which ensures that the thread with ltid$==0$ has sufficient permissions to frame (\sum cont[$*$]) in the barrier postcondition. Notice that when all threads have reached the barrier, all read accesses on region together (including the group resource invariant) can be combined into a write permission on region, thus enabling the update of this ghost variable within the barrier.

Next, we discuss the functional property specifications. As explained before, the two resource invariants specify the values of the shared variables: (1) the local shared variable x must always express the accumulation of the contributions of the threads executing the first atomic operation (line 4), and (2) the global shared variable r must always express the accumulation of x's final value in each work group which is stored in sums[gid] (line 2). To prove these invariants, each thread must ensure that it correctly stores its contribution as specified in line 9. Moreover, the barrier must ensure that the thread with ltid$==0$ knows the final value of x as specified by x==(\sum cont[$*$]) in the barrier's postcondition. Finally, the thread with ltid$==0$ must guarantee that the final value of x is stored in sums[gid] (line 11). Therefore, the verifier can prove that the value of r is the collective contributions of all the threads in the kernel.

6.4. Verification Method and Soundness

The previous section illustrated how we specify resources and functional properties of different kernels in the presence of atomic operations and barriers on several examples. This section presents permission-based separation logic rules for proving data race freedom and functional correctness. We also discuss the soundness of the proof rules.

6.4.1 Verification Method

Given a fully annotated kernel, verification of the kernel with respect to its specification essentially boils down to verification of the following properties:

- Each thread is verified with respect to the thread specification. This means

that given the thread's code T_{body}, the following Hoare triple

$$K_{rinv}, G_{rinv}(gid) \vdash \{T_{pre}\}\ T_{body}\ \{T_{post}\}$$

should be verified using the permission-based separation logic rules defined in Figure 6.3. Each barrier is verified as a method call with precondition B_{pre} and postcondition B_{post}.

- The kernel resources are sufficient for the distribution over the work groups, as specified by the group resources.
- The kernel precondition implies the work group's preconditions.
- The group resources and accesses to local memory are sufficient for the distribution of resources over the threads.
- The work group precondition implies the thread's preconditions.
- Each barrier redistributes only resources that are available in the work group.
- For each barrier the postcondition for each thread follows from the precondition in the thread, and the separating conjunction of the preconditions of all other threads in the work group.
- The universal quantification over all threads' postconditions implies the work group's postcondition.
- The universal quantification over all work groups' postconditions implies the kernel's postcondition.

Figure 6.3 shows the most important proof rules to reason about kernel threads. Rule **[Assign]** describes the updates to the thread's private memory. Rules **[LRead]** and **[LWrite]** specifies read and write of local memory. $L[e]$ denotes the value stored at location e in the local memory array, and substitution is as usually defined for arrays [Apt81]:

$$L[e][L[e_1] := e_2] = (e = e_1)?e_2 : L[e]$$

Similarly the rules **[GRead]** and **[GWrite]** describe read and write of global memory. The rules **[LAtomic]** for local and **[GAtomic]** for global atomic operations are the simple instances of the CSL rule using the group resource invariant and kernel resource invariant, respectively. The rules for sequential composition, conditionals, loops, and weakening are standard and not mentioned in the figure.

The rule **[LBarrier]** reflects the functionality of the barrier. It acts similar to the CSL rule for the group resource invariant but at the same time it collects

resources and knowledge from all threads and redistributes these resources and knowledge. To do so, it requires that the block S can be executed given the resources provided by the invariant (G_{rinv}) and all threads in the work group ($\bigstar_{t\in[0..gsize)} R(t)$). Moreover, it ensures that all resources are given back ($\bigstar_{t\in[0..gsize)} E(t)$) and the invariant is re-established (G_{rinv}). The rule also says that the effect of passing through a barrier on a thread is to give up resources $R(t)$ and get $E(t)$ in return. Note that there is a side-condition that S, R and E can only refer to local memory, as this would otherwise potentially create a data race: a local barrier works as a memory fence on local memory, thus it can only exchange information about local memory locations, but not about the global memory locations. The **[GBarrier]** rule is symmetric in the use of local vs. global memory and invariants. Note that the local or global flag only affects memory accesses and for both flags the barrier can only synchronize the threads within a single work group.

In our verification technique barrier divergence is not taken into consideration. This means that if threads in a work group arrive at a barrier they all arrive at the *same one*. This is a realistic assumption: according to the OpenCL semantics, the behaviour of programs with barrier divergence is unspecified [NVI13]. Therefore, we add some additional syntactical restrictions that ensure that some private variables have the same value in all threads. With this restriction, our kernels do not suffer from *barrier divergence* and we can use these private variables in barrier specifications. Note that the conditions are similar to the checks for control flow uniformity used in the Microsoft C++ AMP compiler [GM12] (See Section 8.1.1 in Microsoft C++ AMP Specification [Mica]). A detailed discussion about barrier divergence in GPGPU kernels and an approach to verify divergence freedom has been presented in [BCD+12].

6.4.2 Soundness

This section discusses the soundness of our verification technique.

Theorem 6.1. *Given a barrier divergence free kernel, for which the thread level Hoare triples are provably correct. Then every possible execution of the kernel starting in a state that satisfies the kernel precondition is data race free and ends in a state that satisfies the kernel postcondition.*

Proof. We are given a finite trace of executions.

Listing 12 PVL translation for the example in Listing 9.

```
1  class ref {
2    invariant a != null ** b != null ** N > 0 **
3                \array(a, N) ** \array(b, N);
4    context (\forall* int i; 0 <= i && i < N; Perm(b[i],write));
5    context (\forall* int i; 0 <= i && i < N; Perm(a[i],write));
6    context (\forall* int i; 0 <= i && i < N; b[i] == (i+1)%N);
7    void shift (int N, int[N] a, int[N] b){
8     par kern(int g = 0 .. 0)
9          context (\forall* int i; 0 <= i && i < N ; Perm(a[i],write));
10    {
11       par workgroup(int t = 0 .. N)
12           requires Perm(a[t], write);
13           ensures Perm(a[t],1/2);
14       {
15         a[ t ] = t;
16         barrier (workgroup)
17           requires Perm(b[t], write) ** Perm(a[t],1/2) ** a[t]==t;
18           ensures Perm(a[t],1/2) ** Perm(a[(t+1) % N],1/2) ** Perm(b[t],write);
19           ensures a[(t+1)%N] == (t+1)%N;
20         {}
21         b[ t ] = a[ (t+1) % N ];
22 }}}}
```

In this trace every thread $t_{gid,ltid}$ makes a finite number of steps $N_{gid,ltid}$, where atomic blocks and barriers count as one step. Because a Hoare logic proof of the thread exists, we can find formulas $P^0_{gid,ltid}, \cdots, P^{N_{gid,ltid}}_{gid,ltid}$ that are valid before, between and after these steps, where $P^0_{gid,ltid}$ is the precondition of the thread and $P^{N_{gid,ltid}}_{gid,ltid}$ is its postcondition.

All states $\sigma_0, \cdots, \sigma_N$ in the finite global trace of N steps can be described by a function f that maps each global trace position to the positions in the local threads. We do not know in which order the steps of the threads are executed, but we know they all start in the position 0, so $f(0, gid, ltid) = 0$. We also know they end in their last state, so: $f(N, gid, ltid) = N_{gid,ltid}$.

We claim that before and after every step in the trace the state satisfies a

specific separation logic formula.

$$\forall i = 0, \cdots, N : \ \sigma_i \models K_{rinv} \star \bigstar_{gid \in [0..ks)} \left(G_{rinv}(gid) \star \bigstar_{ltid \in [0..gsize)} P_{gid,ltid}^{f(i,gid,ltid)} \right)$$

This claim is proven by induction on i. For $i = 0$ this is precisely the given precondition. Assuming that the claim is correct for $0 \leq i < N$, there are three cases. If the step is a plain step or an atomic step, by correctness of the standard CSL Hoare triple used to prove that step, the validity for $i + 1$ follows.

The interesting case is the barrier step, in which all threads of a group are involved. The Hoare triple for each thread is valid so each thread starts knowing $P(t) \star R(t)$ and ends knowing $P(t) \star E(t)$. Because of the correctness of the standard CSL Hoare triple for the barrier statement S, the change to the state is from $\bigstar_{t \in [0..gsize)} R(t) \star G_{rinv}(gid)$ to $\bigstar_{t \in [0..gsize)} E(t) \star G_{rinv}(gid)$, which is precisely the change in the formulas, so $i + 1$ is established.

The last statement is precisely the kernel postcondition which proves that the end state satisfies the kernel postcondition.

A data race happens if: there is an access to a location l in step i_1 by thread t_1, followed by an access to the same location in step i_2 by thread t_2, there is no memory fence in between these accesses, and one of these accesses is a write. Suppose that the thread t_1 used the fraction p_1 for the access and the thread t_2 used the fraction p_2. Because one of the accesses is a write, $p_1 + p_2 > 1$. Because there is no memory fence, that is no barrier or atomic in between, at time i_1 the thread t_2 must have already owned the fraction p_2. Thus at time i_1, the fraction $p_1 + p_2$ permission for location l existed, which leads to a contradiction. □

6.5. Implementation

This section discusses how our verification technique is implemented in the VerCors toolset. First, OpenCL programs are encoded into internal language of VerCors called Prototypal Verification Language (PVL). Second, the generated PVL program is encoded into Viper [JKM+14] and then verified by the Silicon verifier. Viper is an intermediate language for separation logic-like specifications used by Viper project [JKM+14, Vip17, HKMS13, MSS16].

Listing 12, 13 and 16 show the encoding of the OpenCL examples in Listing 9, 10 an 11, respectively. The kernels are encoded into two nested parallel blocks in PVL where the parallel block kern (lines 8-22 in Listing 12) encodes

Listing 13 PVL translation for the example in Listing 10.

```
1  class Ref {
2    invariant values != null ** temp != null ** N > 0 ** \array(values, N)
3                ** \array(x, write);
4    context    (\forall* int i; 0 <= i && i < N; Perm(values[i],1/2));
5    context    Perm(x[0], write);
6    void do_sum(int N, int[N] values, int[1] x)
7    {
8        x[ 0 ] = 0;
9        invariant inner(Perm(x[0], write)
10                   ** (\forall* int j; 0 <= j && j < N; Perm(values[j],1/2)))
11       {
12           par workgroup(int t = 0 .. N)
13               context Perm(values[t],1/2);
14           { atomic(inner){ x[0] = x[0] + values[t]; } }
15       }
16 }}
```

the work groups. The parallel block workgroup (lines 11-22 in the same Listing) encodes the threads in a particular work group. The kernel in Listing 12 uses only one single work group because barriers can only synchronize the threads of a single work group. All variables defined outside of the kernel parallel block kern are taken as global variables, while all variables inside are assumed to be local variables. The variables defined inside the parallel block workgroup are assumed to be thread private variables.

Listing 13 shows how atomic operations, in this case OpenCL's atomic_add operation, are encoded into an atomic block (line 14 in Listing 13 and lines 21, 30, and 31 in Listing 16). The kernel and group resource invariants are encoded into PVL's invariant block invariant(ϕ)$\{\dots\}$ where ϕ is the invariant formula and $\{\dots\}$ is the block of code in which the invariant holds. The PVL translation in Listing 13 uses one inner invariant block which captures the group resource invariant while the example in Listing 16 needs two resource invariants outer and inner to capture both kernel and group resource invariants respectively.

In Listing 16, the variable res is defined in global memory; because it is accessible to all threads in all work groups. The properties over res is specified

Listing 14 Vectorization of a loop with a forward loop-carried dependence

```
for (int j=0; j < N; j++)                          requires φ(tid);
  requires φ(j);                                   ensures ψ(tid);
  ensures ψ(j);                                    __kernel loop(...)
{                                                  {
  L_1: if (g_1(j)) { I_1(j); }          ⇒              C_1(tid);
   ⋮                                                  ⋮
  L_K: if (g_K(j)) { I_K(j); }                     C_K(tid);
}                                                  }
```

by the kernel resource invariant. Similarly, temp is stored in local memory, which is accessible only to the threads in the same work group, so the properties over temp are specified in the group resource invariant. For brevity, Listing 13 and 16 shows only the specifications required for reasoning about the data race freedom of the kernel examples.

As discussed before in Section 6.4, to verify a kernel our method gives rise to the following proof obligations:

1. several global properties to ensure that the correct relation between different levels of specifications (e.g. all kernel resources are properly distributed over a work group, and the universally quantified barrier precondition implies the universally quantified barrier postcondition);
2. the correctness of a single arbitrary thread with respect to its specifications; and
3. ensuring the correct framing of each pre- and postcondition.

To verify the PVL translation of a kernel, the above-mentioned proof obligations are encoded into Viper. To do so, for each proof obligation of the form "ϕ implies ψ", an annotated Viper method with precondition ϕ and postcondition ψ and an empty method body is generated. Then the encoded proof obligation is verified by passing the annotated method to the Silicon verifier [JKM+14, HKMS13, MSS16, Vip17]. For example, consider the kernel in Listing 9 and its PVL translation in Listing 12. It is executed with a single work group setting; therefore the only work group has exactly the same resources as the kernel. To verify if the group resources are properly distributed over the threads at the barrier, the following Viper encoding is generated:

```
requires (\forall* int tid; 0<=tid && tid<gsize;
          acc(getVCTOption(a)[tid].Integer_item, write));
requires (\forall* int tid; 0<=tid && tid<gsize;
          acc(getVCTOption(b)[tid].Integer_item, write));
ensures  (\forall* int tid; 0<=tid && tid<gsize;
          acc(getVCTOption(a)[(tid+1)%gsize].Integer_item,1/2));
ensures  (\forall* int tid; 0<=tid && tid<gsize;
          acc(getVCTOption(b)[tid].Integer_item,1/2));
void main_resources(int tcount, int gsize, int gid){}
```

where acc is the keyword for permission predicates in the Viper intermediate language, this is equivalent to the Perm keyword in our notation. The function getVCTOption is the specific support to encode arrays into Viper sequences. Arrays are not present as a separate primitive in Viper; so special treatment is required to deal with them. The implementation of this encoding is available in the open source version of our toolset [Ver17b].

6.6. Compiling Iteration Contracts to Kernel Specifications

In Chapter 3 we discussed verification of loop parallelizability in high-level sequential programs. Typically, we want to be sure that when we parallelize the program, the resulting low-level parallel code is still correct. To support this, we define how a specification of the original program can be translated into a specification of the low-level code. In particular, this section shows how iteration contracts are translated into OpenCL kernel specifications, such that if the code is compiled using a basic parallelizing compiler, without further optimization, the compiled code is correct with respect to the compiled specification.

Independent Loops. Given an independent loop, the basic compilation to kernel code is simple: create a kernel with as many threads as there are loop iterations and each kernel thread executes one iteration. Moreover, the iteration contract can be used as the thread contract for each parallel thread in the kernel directly. The size of the work group can be chosen at will, because no barriers are used.

Forward Loop-carried Dependencies If the loop has forward dependencies then the kernel must mimic the vectorized execution of the loop. Consider the specified loop on the left side of Listing 14, for simplicity, we assume that both the number of threads and the size of the work group are N. Basic vectorization generates the kernel on the right side of the listing, where:

- if $I_k(j)$ is a send statement then it is ignored: $C_k(j) \equiv \{\}$
- if $I_k(j)$ is a recv statement with a matching send statement at L_i, then it is replaced by a barrier $C_k(j) \equiv$

  barrier $(\ldots)$ requires $g_i(j) \Rightarrow \phi_S(j)$; ensures $g_k(j) \Rightarrow \phi_R(j)$; $\{\}$

  where the barrier contract specifies how the permissions are exchanged at the barrier. The barrier flag in the generated kernel can be set to the combination of local and global memory flags.
- if $I_k(j)$ is any other statement then it is copied:

$$C_k(j) \equiv \text{if } (g_k(j))\{\ I_k(j);\ \}$$

Listing 15 shows the kernel that is derived in this way from the forward dependence example in Listing 2(a).

6.7. Related Work

Bardsley et al. propose additional support in GPUVerify [BCD+12] for reasoning about GPU kernels where warps and atomic operations are used for synchronization [BD14]. In GPUVerify the user does not need to manually add specifications, because the tool internally speculates and refines kernel specifications [BCD+12]. However, GPUVerify is not able to reason about the functional properties of kernels, it can only prove the absence of data races. As future work, we would like to investigate if GPUVerify could be used to infer some of the annotations that we need.

Concerning verification of GPU kernels, we should also mention the work of Li and Gopalakrishnan [LG10]. They verify CUDA programs by symbolically encoding thread interleavings. They were the first to observe that to ensure data race freedom it was sufficient to verify the interleavings of two arbitrary threads. For each shared variable they use an array to keep track of read and

Listing 15 Kernel implementing the loop with forward dependence

```
1  /*@
2    requires Perm(a[tid], write) ** Perm(b[tid],1/2) ** Perm(c[tid], write)
3             ** b[tid]==tid;
4    ensures  Perm(a[tid],1/2) ** Perm(b[tid],1/2) ** Perm(c[tid], write);
5    ensures  (tid>0) ==> Perm(a[tid-1],1/2);
6    ensures  (tid==tcount-1) ==> Perm(a[tid],1/2);
7    ensures  a[tid]==tid+1 ** b[tid]==tid ** ((tid>0) ==> c[tid]==tid+2);
8  @*/
9  __kernel void ForwardDepKernel(__global int * a, __global int * b,
10                 __global int * c)
11 {
12     a[ tid ] = b[ tid ] + 1;
13     barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE)
14     /*@
15       requires (tid<tcount-1) ==> Perm(a[tid],1/2) ** a[tid]==tid+1;
16       ensures  (tid>0) ==> Perm(a[tid-1],1/2) ** (tid>0) ==>a[tid-1]==tid;
17     @*/{}
18     if (tid>0)   c[ tid ] = a[ tid-1 ] + 2;
19 }
```

write accesses, and where in the code they occur. By analyzing this array, they detect possible data races. However, they do not consider atomic operations.

Chiang et al. present a formal bug-hunting method for GPU kernels [CGLR13]. Their method extends the GKLEE tool [LLS+12] with the analysis of CUDA kernels that use both barriers and atomics. To detect a potential conflict involving atomic accesses, thread schedules are enumerated in order to find a counterexample to correctness. Delay bounding is used to limit schedule explosion. Although their method is effective in finding defects, it cannot be used to verify the absence of data race; while our approach is able to prove race freedom of GPU kernels. Moreover, they do not support verification of functional properties for the GPU kernels that use both barriers and atomic operations.

In the verification of (general) concurrent programs synchronized with barriers, Hobor *et al.* [HG11] propose a sound extension of CSL for pthreads-style barriers. The simplicity of the OpenCL barriers makes our specification simpler. Additionally, we support barriers in the presence of atomic operations.

Listing 16 PVL translation for the example in Listing 11.

```
1  class Ref {
2    int r;
3    invariant values != null ** N > 0 ** M > 0 ** \array(values,M*N);
4    context Perm(r,write)
5              ** (\forall* int i; 0 <= i && i < M*N; Perm(values[i],1/2));
6    requires r==0;
7    void do_sum(int M,int N, int[M*N] values){
8      invariant outer(Perm(r,write))
9      {
10       par kern(int g = 0 .. M)
11           context (\forall* int k; 0 <= k && k < N; Perm(values[g*N+k],1/2));
12       {
13           int[1] x = new int[1]; x[ 0 ] = 0;
14           invariant inner(\array(x,1) ** Perm(x[0],write))
15           {
16               par workgroup(int t = 0 .. N)
17                   requires Perm(values[g*N+t],1/2);
18                   ensures (t == 0) ⇒
19                           (\forall* int k; 0 <= k && k < N; Perm(values[g*N+k],1/2));
20               {
21                   atomic(inner){ x[ 0 ] = x[ 0 ] + values[ g*N+t ]; }
22                   barrier (workgroup)
23                     requires Perm(values[g*N+t],1/2);
24                     ensures (t == 0) ⇒
25                           (\forall* int k; 0 <= k && k < N; Perm(values[g*N+k],1/2));
26                   {}
27                   if (t==0)
28                   {
29                       int tmp;
30                       atomic(inner){ tmp = x[ 0 ]; }
31                       atomic(outer){ r = r + tmp; }
32                     }
33 }}}}}}
```

6.8. Conclusion and Future Work

This chapter presents an approach to specify and verify GPGPU programs in the presence of atomic operations and barriers. The main characteristics of the approach are that it can be used to prove both data race freedom and functional correctness. To specify the shared memory accesses, the notion of resource invariant from CSL is lifted to the GPU memory model, distinguishing

between kernel and group resource invariants. An appropriate Hoare logic is proposed and proven sound to reason about GPGPU programs using atomic operations and barriers. The approach is illustrated on some examples, and supported by an implementation in the VerCors toolset. Moreover, we discuss how the iteration contracts can be translated into kernel specifications. This is specifically useful in automatic loop parallelization where ensuring the semantical equivalence of the original parallel loops and the generated GPGPU kernels is essential.

CHAPTER 7

CONCLUSION

"When done well, software is invisible."

– Bjarne Stroustrup

The current trend in high-performance computing is to exploit the enormous computing power provided by modern multi-core and many-core processors. At software level this can only be achieved by writing parallel programs. As a result, nowadays parallel programming is omnipresent in almost any scientific and business application, even in critical systems where the smallest errors might endanger human life or cause substantial economic damages. However, the hindering challenge is that, in contrast to sequential programs, parallel programs are notoriously error-prone. In this thesis we tackled this challenge by developing novel axiomatic verification techniques that make parallel programming safer and more reliable. Specifically we developed verification techniques based on permission-based separation logic to reason about data race freedom and functional correctness of parallel loops, deterministic parallel programs, and GPGPU kernels. We also demonstrated the practical applicability of the developed techniques by implementing them as part of our VerCors toolset. At the end, we briefly discussed how the verification techniques of this thesis are connected by presenting how iteration contracts can be translated into kernel specifications. This chapter presents the summary of the main contributions of this thesis and ends with discussing future research directions.

7.1. Verification of Loop Parallelization

The first main contribution of this thesis is a technique for reasoning about loop parallelization. With this technique we are able to prove whether a loop that is claimed to be parallel is indeed parallelizable. For this purpose, we developed the notion of iteration contract that specifies the resources (i.e. the memory locations read and written) that are accessed by each iteration of the loop. To prove that a loop is parallelizable we show that its iteration contracts are disjoint, i.e. each iteration accesses to a separated part of memory.

Next we discussed how the technique supports the specification of loop-carried data dependencies by extending iteration contracts with extra annotations. The annotations capture how resources are transferred from one iteration to another iteration of the loop. The loops with loop-carried data dependencies are either inherently sequential (in the case of a backward loop-carried dependency) or they can be parallelized if they are properly synchronized (in the case of a forward loop-carried dependency). The technique is able to distinguish between either of these cases and in the latter case, the extra annotations indicate where in the loop the additional synchronization should be added.

Moreover, iteration contracts can be extended further such that they specify the functional behaviour of each iteration. This allows to seamlessly verify the functional correctness of the parallel loop together with its parallelizability. Finally we explained how the technique is implemented as part of our VerCors toolset.

7.2. Reasoning about Deterministic Parallel Programs

As the second main contribution of the thesis, in Chapters 4 and 5 we presented a novel technique for the verification of deterministic parallel programs. First we captured the core features of deterministic parallel programming in the Parallel Programming Language (PPL). PPL is a language for the composition of code blocks. We distinguished between three kinds of basic blocks: a parallel block, a vectorized block and a sequential block. Basic blocks are iterative blocks; assuming that the sequential basic block only has one iteration. Basic blocks can be composed by three binary block composition operators: sequential composition, parallel composition and fusion composition. A small-step operational semantics was presented for PPL.

A PPL program defines a partial ordering over its basic blocks from which a partial order over the iterations can be inferred. From these partial orderings a set of independent (incomparable) iterations is statically computed. We showed that if each basic block is data race free and all incomparable iterations are non-conflicting (i.e. they access to disjoint memory partitions) then the PPL program is data race free.

Next we discussed how iteration contracts extended with functional specifications are used to reason about the functional correctness of the PPL program. For this purpose, we first showed that if the program is data race free, it can

be linearized; thus the functional correctness of a data race free program can be proven over its linearized variant, to which the standard separation logic for sequential programs is applicable.

Finally we illustrated the practical applicability of the technique by discussing how a commonly used subset of OpenMP can be encoded into PPL and then verified. The complete process from the encoding to the verification is implemented as part of the VerCors toolset.

7.3. Verification of GPGPU Programs

As the third main contribution, a previous technique for the verification of GP-GPU programs was extended to support atomic operations. To be able to specify the shared memory accesses of atomic operations, the notion of resource invariant is lifted to the GPU memory model. We distinguish between kernel and group resource invariants. An appropriate Hoare logic is proposed and proven sound to reason about GPGPU programs using atomic operations and barriers. The approach is illustrated on some examples and prototyped in our VerCors toolset.

In Section 6.6 we also showed how iteration contracts can be translated into kernel specifications. This is especially interesting when parallel loops as a high-level parallel program are transformed into kernels as a low-level parallel program. Therefore, specifications may be provided once only on the high-level program and then can be reused for the low-level kernel via translation. In this way we are able to not only reason about the correctness of parallel loops and kernels but also to reason about the correctness of the parallelization framework as a whole; including all intermediate transformations.

7.4. Future Work

This section summarizes future research directions.

Verifiers and Parallelizing Compilers. As discussed previously in Chapter 3, there are connections between verifiers and parallelizing compilers. One of the possible future directions is to investigate how verifiers and parallelizing compilers can support each other. We believe this support can work in both ways. First of all, parallelizing compilers can use verified annotations to

know about dependencies without analyzing the code itself. Conversely, if the compiler performs an analysis then it could emit its findings as a specification template for the code, from which a complete specification can be constructed. This might be extended to a set of techniques for automatic generation of iteration contracts. Moreover, some techniques in parallelizing compilers (e.g. polyhedral compilation [Bas04, BPCB10]) perform specific loop transformations in order to make loops more suitable for parallelization. Thus it could be studied further how iteration contracts can be in a similar way transformed such that they are still valid for the transformed loop.

Fully Automatic Verification of Deterministic Parallel Programs. In Chapter 5, we showed that for PPL programs to be verified we only need to provide a program contract and an iteration contract for each basic block in the program. So the burden of specifying all block compositions has already been removed. As a next step towards more automated verification technique, the iteration contracts (more specifically the resource formula part of the contracts) can be generated automatically. If it is achieved, data race freedom of deterministic parallel programs can be fully automatically verified. We believe that we took the first steps towards this goal, by presenting an instrumented operational semantics for PPL in Section 5.2. This provides the required formalism to calculate the memory masks in different program states from which the permission fractions of the resource formulas can be extracted.

Extensions of PPL. Our verification technique for deterministic parallel programs is presented over the PPL language. The language covers the core features of deterministic parallel programming. As the next step, PPL can be extended to address more deterministic parallel programming features. In particular, OpenMP task constructs and atomic operations (when they comply with reduction patterns) can be considered.

Chaining the Verification Techniques. The verification techniques presented in this thesis can be used independently. Moreover, they can also be chained together to be used as a holistic verification solution for a complete parallelization framework where programs are written in a high-level parallel programming language with native parallelization constructs such as OpenMP and then automatically compiled into low-level parallel programs such as GPU kernels or multi-threaded programs. To make this happen, the specifications

in the high-level language should also be translated automatically into the specification of the low-level language. In Section 6.6 we took the first step in this direction by demonstrating how iteration contracts can be translated into kernel specifications. This can be explored even further for cases where the low-level programming language is a multi-threaded program or when the high-level parallel program is a deterministic parallel program that uses a diverse set of parallelization constructs. This holistic verification solution is particularly interesting because it checks not only the correctness of high-level program and its low-level counterpart, but also it verifies the correctness of the intermediate transformations by checking the semantics equivalence of the high-level and low-level programs.

List of Publications by the Author

[1] Saeed Darabi, Stefan C. C. Blom, and Marieke Huisman. A Verification Technique for Deterministic Parallel Programs. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 247–264, Cham, 2017. Springer International Publishing.

[2] Saeed Darabi, Stefan Blom, and Marieke Huisman. *A Verification Technique for Deterministic Parallel Programs (extended version)*. Number TR-CTIT-17-01 in CTIT technical report. Centre for Telematics and Information Technology (CTIT), Netherlands, 2017.

[3] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods: 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, pages 102–110, Cham, 2017. Springer International Publishing.

[4] Afshin Amighi, Saeed Darabi, Stefan Blom, and Marieke Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 69–83, Cham, 2015. Springer International Publishing.

[5] Stefan Blom, Saeed Darabi, and Marieke Huisman. Verification of Loop Parallelisations. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering: 18th International Conference,*

FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings, pages 202–217. Springer Berlin Heidelberg, 2015.

[6] Stefan Blom, Saeed Darabi, and Marieke Huisman. Verifying Parallel Loops with Separation Logic. In Alastair F. Donaldson and Vasco T. Vasconcelos, editors, Proceedings 7th Workshop on *Programming Language Approaches to Concurrency and Communication-cEntric Software*, Grenoble, France, 12 April 2014, volume 155 of *Electronic Proceedings in Theoretical Computer Science*, pages 47–53. Open Publishing Association, 2014.

[7] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva. Verification of Concurrent Systems with VerCors. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 8483 in Lecture Notes in Computer Science, pages 172–216. Springer, 6 2014.

References

[ADBH15] A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In *Software Engineering and Formal Methods*, pages 69–83. Springer, 2015.

[AF11] A. Aviram and B. Ford. Deterministic OpenMP for Race-free Parallelism. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism,HotPar'11*, pages 4–4, 2011.

[AMD] AMD Corporation. White Paper — AMD Graphics Cores Next (GCN) Architecture. `https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[Apt83] K. R. Apt. Ten years of Hoare's Logic: a survey—Part II: nondeterminism. *Theoretical Computer Science*, 28(1-2):83–109, 1983.

[ari17] Stanford Encyclopedia of Philosophy, Last accessed Sep. 25, 2017. `https://plato.stanford.edu/entries/aristotle-logic/`.

[BAAS09] R.L. Bocchino, V.S. Adve, S.V. Adve, and M. Snir. Parallel Programming Must be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 4–4, 2009.

[BAMM05] M. J. Berger, M. J. Aftosmis, D. D. Marshall, and S. M. Murman. Performance of a New CFD Flow Solver Using a Hybrid Programming Paradigm. *Journal of Parallel Distributed Computing*, 65(4):414–423, 2005.

[Bas04] C. Bastoul. Code Generation in the Polyhedral Model is Easier than You Think. In *Proceedings of the 13th International Conference*

on Parallel Architectures and Compilation Techniques, pages 7–16. IEEE Computer Society, 2004.

[BBC+15] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. Donaldson, and J. Ketema. PENCIL: A Platform-neutral Compute Intermediate Language for Accelerator Programming. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 138–149. IEEE, 2015.

[BCD+12] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 113–132. ACM, 2012.

[BCD+15] A. Betts, N Chong, A. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(3):10, 2015.

[BCG+13] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From Relational Verification to SIMD Loop Synthesis. In *ACM SIGPLAN Notices*, volume 48, pages 123–134, 2013.

[BCG+15] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, and A. Donaldson. PENCIL 1.0 Language Specification. Research Report 8706, INRIA, 2015.

[BCO05] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission Accounting in Separation Logic. In *Principles of Programming Languages*, pages 259–270, 2005.

[BCY06] R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[BD14] E. Bardsley and A. Donaldson. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *NASA Formal Methods*, volume 8430 of *LNCS*, pages 230–245. Springer, 2014.

[BDH] S. C. C. Blom, S. Darabi, and M. Huisman. In *Fundamental Approaches to Software Engineering (FASE)*, volume 9033 of *LNCS*. Springer.

[BDH14] S.C.C. Blom, S. Darabi, and M. Huisman. Verifying Parallel Loops with Separation Logic. In Proceedings 7th Workshop on *Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES)*, pages 47–53, 2014.

[BDHO17] S.C.C. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods: 13th International Conference (IFM)*, pages 102–110. Springer, 2017.

[BDJ12] M. Botincan, M. Dodds, and S. Jagannathan. Resource-sensitive Synchronization Inference by Abduction. In *Principles of Programming Languages (POPL)*, pages 309–322, 2012.

[BDJ13] M. Botinčan, M. Dodds, and S. Jagannathan. Proof-Directed Parallelization Synthesis by Separation Logic. *ACM Transactions on Programming Languages and Systems*, 35:1–60, 2013.

[BH14] S.C.C. Blom and M. Huisman. The VerCors tool for Verification of Concurrent Programs. In *Formal Methods (FM)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[BHM14] S. C. C. Blom, M. Huisman, and M. Mihelčić. Specification and Verification of GPGPU programs. *Science of Computer Programming*, 95(Part 3):376 – 388, 2014.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and control*, 60(1-3):109–137, 1984.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In *Static Analysis Symposium*, LNCS, pages 55–72. Springer, 2003.

[BPCB10] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model is More Widely Applicable Than You Think.

In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303. Springer, 2010.

[Bur72] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972.

[CAR17] CARP Project, Last accessed Sep. 25, 2017. `http://carp.doc.ic.ac.uk/external/`.

[CGLR13] W. Chiang, G. Gopalakrishnan, G. Li, and Z. Rakamarić. Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-bounding. In *NASA Formal Methods Symposium*, pages 213–228. Springer, 2013.

[CK11] B. Cowan and B. Kapralos. GPU-based Acoustical Occlusion Modeling with Acoustical Texture Maps. In *Proceedings of the 6th Audio Mostly Conference: A Conference on Interaction with Sound*, AM '11, pages 55–61. ACM, 2011.

[cli17] CilkPlus. accessed Nov. 28, 2017. `https://www.cilkplus.org/`.

[DBH17a] S. Darabi, S. C. C. Blom, and M. Huisman. A Verification Technique for Deterministic Parallel Programs. In *NASA Formal Methods: 9th International Symposium (NFM) Proceedings*, pages 247–264. Springer, 2017.

[DBH17b] S. Darabi, S.C.C. Blom, and M. Huisman. A verification technique for deterministic parallel programs (extended version). Technical Report TR-CTIT-17-01, Enschede, the Netherlands, February 2017.

[Dij72] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1972.

[DJP11] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular Reasoning for Deterministic Parallelism. In *ACM SIGPLAN Notices*, pages 259–270, 2011.

[DMB08] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[Dow97] M. Dowson. The Ariane 5 Software Failure. *SIGSOFT Software Engineering Notes*, 22(2):84–, 1997.

[Flo93] R. W. Floyd. Assigning Meanings to Programs. In *Program Verification*, pages 65–81. Springer, 1993.

[GH06] T. Gedell and R. Hähnle. Automating Verification of Loops by Parallelization. In *Logic for Programming Artificial Intelligence and Reasoning*, pages 332–346, 2006.

[GM12] K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Developer Reference Series. Microsoft Press, 2012.

[Heh05] E. C. R. Hehner. Specified Blocks. In *Verified Software: Theories, Tools, Experiements*, pages 384–391, 2005.

[HG11] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic. In *20th European Symposium of Programming (ESOP 2011)*, LNCS, pages 276–296. Springer, 2011.

[HHH08] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008.

[HHHA14] C. Haack, M. Huisman, C. Hurlin, and A. Amighi. Permission-based Separation Logic for Multithreaded Java Programs. *Logical Methods in Computer Science*, 2014.

[HHI+01] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. On the Unusual Effectiveness of Logic in Computer Science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[HKMS13] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 451–476. Springer, 2013.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HSP15] D. D. Hoffman, M. Singh, and C. Prakash. The Interface Theory of Perception. *Psychonomic Bulletin & Review*, 22(6):1480–1506, Dec 2015.

[Hur09a] C. Hurlin. Automatic Parallelization and Optimization of Programs by Proof Rewriting. In *Static Analysis Symposium*, pages 52–68. Springer, 2009.

[Hur09b] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.

[HW73] C. Antony R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[IO01] S. S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. *ACM SIGPLAN Notices*, 36(3):14–26, 2001.

[JFY99] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance, 1999.

[JKM+14] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zurich, 2014.

[JM97] J-M Jazequel and B. Meyer. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, 1997.

[JP11] B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. *ACM SIGPLAN Notices*, 46(1):271–282, 2011.

[JS10] Edward K. Jason S. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[Khr] Khronos OpenCL Working Group. The OpenCL Specifications - Atomic Functions. `https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/atomicFunctions.html`.

[Kni02] J. C. Knight. Safety Critical Systems: Challenges and Directions. In *Proceedings of the 24th International Conference on Software Engineering*, International Conference on Software Engineering, pages 547–550. ACM, 2002.

[L+96] M. R. Lyu et al. Handbook of Software Reliability Engineering. 1996.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[Lee06] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[LG10] G. Li and G. Gopalakrishnan. Scalable SMT-based Verification of GPU Kernel Functions. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 187–196. ACM, 2010.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof Rules for the Programming Language Euclid. *Acta Informatica*, 10(1):1–26, 1978.

[LLS+12] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *ACM SIGPLAN Notices*, volume 47, pages 215–224. ACM, 2012.

[LS11] L. Lu and M. Scott. Toward a Formal Semantic Framework for Deterministic Parallel Programming. *Distributed Computing*, pages 460–474, 2011.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[McC61] J. McCarthy. A Basis for a Mathematical Theory of Computation, Preliminary Report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238. ACM, 1961.

[Mica] Microsoft Corporation. The OpenCL Specification. https://blogs.msdn.microsoft.com/nativeconcurrency/2012/02/03/c-amp-open-spec-published/.

[Micb] Microsoft TPL. http://msdn.microsoft.com/enus/library/dd460717.aspx.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[Moo98] G. E. Moore. Cramming More Components onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. In *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer, 2016.

[Mul12] J. B. Mulligan. A GPU-accelerated Software Eye Tracking System. pages 265–268. ACM, 2012.

[Mus80] J. D. Musa. Software Reliability Measurement. 1980.

[NVI] NVIDIA Corporation. Fermi White Paper, NVIDIA's Next Generation CUDA Compute Architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[NVI13] NVIDIA Corporation. CUDA C programming guide, version 5.5, 2013.

[O'H04] P. W. O'Hearn. Resources, Concurrency, and Local Reasoning. In *European Symposium on Programming*, pages 1–2. Springer, 2004.

[O'H07] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[O'H08] P. W. O'Hearn. Separation Logic Tutorial. In *International Conference on Logic Programming*, pages 15–21. Springer, 2008.

[Ope17a] Standard Performance Evaluation Corporation (SPEC), SPEC OMP2001. Last accessed Nov. 28, 2017. www.spec.org/hpg/omp2001.

[ope17b] OpenACC, Last accessed Sep. 25, 2017. `https://www.openacc.org/`.

[ope17c] The OpenCL Specification, Last accessed Nov. 28, 2017. `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html`.

[ope17d] OpenMP Architecture Review Board, OpenMP API Specification for Parallel Programming, Last accessed Nov. 28, 2017. `http://openmp.org/wp/`.

[Ope17e] LLNL OpenMP Benchmarks, Last accessed Nov. 28, 2017. `https://asc.llnl.gov/CORAL-benchmarks/`.

[OR12] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. *SIGPLAN Not.*, 47(6):509–520, 2012.

[ORY01] P. W. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Computer science logic*, pages 1–19. Springer, 2001.

[Par17] Parallel Loops, Last accessed Sep. 25, 2017. `https://msdn.microsoft.com/en-us/library/ff963552.aspx`.

[PS11] M. J. Parkinson and A. J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. Springer, 2011.

[RB04] F. J. L. Reid and J. M. Bull. OpenMP Microbenchmarks Version 2.0. *In European Workshop on OpenMP*, 2004.

[RD13] C. Radoi and D. Dig. Practical Static Race Detection for Java Parallel Loops. pages 178–190, 2013.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[ROA12] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83. IEEE Computer Society, 2012.

[Rob12] A. D. Robison. Cilk Plus: Language Support for Thread and Vector Parallelism. *Talk at HP-CAST*, 18:25, 2012.

[RPR07] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity Analysis for Automatic Parallelization on Multi-cores. In *Proceedings of the 21st Annual International Conference on Supercomputing*, pages 263–273. ACM, 2007.

[RVY13] V. Raychev, M. Vechev, and E. Yahav. Automatic Synthesis of Deterministic Concurrency. In *Static Analysis Symposium*, pages 283–303. Springer, 2013.

[SEU+15] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin. The Shift to Multicores in Real-time and Safety-critical Systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 220–229, 2015.

[SHT+08] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, Z. Liang, and B. P. Sutton. Accelerating Advanced MRI Reconstructions on GPUs. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 261–272. ACM, 2008.

[SJP12] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *ACM Transactions on Programming Languages and Systems*, 34(1):2:1–2:58, 2012.

[SMA14] J. Salamanca, L. Mattos, and G. Araujo. Loop-Carried Dependence Verification in OpenMP. In *Using and Improving OpenMP for Devices, Tasks, and More (IWOMP)*, pages 87–102, 2014.

[SRS+12] J. A Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, Geng D. Liu, and W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

[Sta17] State of the Lambda: Libraries Edition., Last accessed Nov. 25, 2017. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html`.

[TC13] S. Tsutsui and P. Collet. *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 2013. Understanding NVIDIA GPGPU Hardware, pages 14-34.

[Thr] Threading Building Blocks. `http://threadingbuildingblocks.org`.

[Vaf11] V. Vafeiadis. Concurrent Separation Logic and Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.

[VCJC+13] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23, 2013.

[Ver17a] VerCors Project, Last accessed Nov. 28, 2017. `http://www.utwente.nl/vercors/`.

[Ver17b] VerCors Toolset, Last accessed Nov. 28, 2017. `https://github.com/utwente-fmt/vercors`.

[Vip17] Viper Project, Last accessed Nov. 28, 2017. `http://www.pm.inf.ethz.ch/research/viper`.

Summary

This thesis presents novel formal verification techniques to improve the reliability of parallel programs and to prove their functional correctness. For this purpose we use axiomatic reasoning techniques based on permission-based separation logic. Among different parallel programming paradigms, we specifically focus on the deterministic parallel programming where the parallelization is expressed over a sequential program using high-level parallelization constructs (e.g. parallel loops).

After a brief introduction to permission-based separation logic in Chapter 2, Chapter 3 presents a verification technique to reason about loop parallelizations. We introduce the notion of an *iteration contract* that specifies the memory locations being read or written by each iteration of the loop. The specifications can be extended with extra annotations that capture loop-carried data-dependencies and functional behavior of the loop. A correctly written iteration contract can be used to draw conclusions about the functional correctness and the safety of a loop parallelization.

Chapter 5 presents a novel technique to reason about deterministic parallel programs. To do that, we first in Chapter 4 formally define the *Parallel Programming Language* (PPL) as a core language that captures the main forms of deterministic parallel programs. This language distinguishes between three kinds of basic blocks: parallel, vectorized and sequential blocks, which can be composed using three different composition operators: sequential, parallel and fusion composition. We show that it is sufficient to have contracts for the basic blocks to prove the correctness of PPL programs, and moreover when a PPL program is data race free, the functional correctness of the sequential program implies the correctness of the parallelized program.

In Chapter 6, we propose a specification and verification technique to reason about the data race freedom and functional correctness of GPGPU kernels that use atomic operations as a synchronization mechanism. For this purpose we adapt the notion of resource invariant to the GPGPU memory model such that

group resource invariants capture the behaviour of atomic operations that access locations in local memory while *kernel resource invariants* capture the behaviour of atomic operations that access locations in global memory. Finally, Chapter 7 concludes the thesis and presents some future directions.

Samenvatting

Dit proefschrift presenteert nieuwe formele verificatietechnieken om de betrouwbaarheid van parallelle programma's te verbeteren, en hun functionele correctheid te bewijzen. Voor dit doel gebruiken we axiomatische redeneringstechnieken, gebaseerd op permissie-gebaseerde separatie logica. We kijken in het bijzonder naar deterministisch parallel programmeren, waarbij de parallellisatie van een sequentieel programma wordt gekarakteriseerd door hoogniveau parallellisatie constructies (bijvoorbeeld parallelle lussen).

Na een korte introducie van permissie-gebaseerde separatie logica in Hoofdstuk 2, presenteert Hoofdstuk 3 een verificatietechniek om te redeneren over lusparallellisaties. We introduceren het begrip 'iteratiecontract', om te specificeren welke geheugenlocaties geschreven of gelezen worden voor elke iteratie van de lus. De specificaties kunnen uitgebreid worden met extra annotaties die zowel lus-gerelateerde data-afhankelijkheden karakteriseren, als functionele eigenschappen van de lus. Een correct iteratiecontract kan gebruikt worden om conclusies te trekken over de functionele correctheid van de lus, en over de veiligheid van de lusparallellisatie.

Hoofdstuk 5 beschrijft een nieuwe techniek om over deterministische parallelle programma's te redeneren. Daarvoor introduceren we in Hoofdstuk 4 de taal PPL (Parallel Programming Language) als een kerntaal die de belangrijkste vormen van deterministische parallelle programma's karakteriseren. Deze taal onderscheidt drie soorten basisblokken: parallelle, gevectoriseerde en sequentiële blokken, die gecombineerd kunnen worden met behulp van drie verschillende compositie-operatoren: sequentiële, parallelle en fusie compositie. We laten zien dat het voldoende is om contracten te geven voor de verschillende basisblokken om correctheid van een PPL programma te bewijzen. Boven, als een PPL programma geen data races bevat, dan zal de functionele correctheid van het sequentiële programma de correctheid van het geparallelliseerde programma impliceren.

In Hoofdstuk 6 presenteren we een specificatie- en verificatietechniek om

te redeneren over afwezigheid van data races en functionele correctheid van GPGPU kernels die atomaire operaties gebruiken als synchronisatiemechanisme. Voor dit doel passen we het begrip 'resource invariant' aan aan het GPGPU geheugen model, zodat resource invarianten voor groepen het gedrag van atomaire operaties op het lokale geheugen beschrijven, terwijl kernel resource invarianten het gedrag van atomaire operaties op het globale geheugen beschrijven.

Hoofdstuk 7 tenslotte, concludeert dit proefschrift en beschrijft een aantal richtingen voor toekomstig onderzoek.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software*

Systems. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of

Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an ex-*

ploration with MetaMod. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

This thesis presents a set of verification techniques based on permission-based separation logic to reason about the data race freedom and functional correctness of program parallelizations. Our reasoning techniques address different forms of high-level and low-level parallelization including parallel loops, deterministic parallel programs (e.g. OpenMP) and GPGPU kernels.

Moreover, we discuss how the presented techniques are chained together to verify the semantic equivalence of high-level parallel programs and their low-level counterparts.