Ph.D. DISSERTATION

Reconciling Low-Level Features of C with Compiler Optimizations

C의 저수준 기능과 컴파일러 최적화 조화시키기

BY

Jeehoon Kang

FEBRUARY 2019

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Reconciling Low-Level Features of C with Compiler Optimizations

C의 저수준 기능과 컴파일러 최적화 조화시키기

BY

Jeehoon Kang

FEBRUARY 2019

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Reconciling Low-Level Features of C with Compiler Optimizations

C의 저수준 기능과 컴파일러 최적화 조화시키기

지도교수 허 충 길

이 논문을 공학박사 학위논문으로 제출함

2019 년 1 월

서울대학교 대학원

전기 컴퓨터 공학부

강 지 훈

Jeehoon Kang의 공학박사 학위논문을 인준함

2019 년 1 월

위 원 장	이광근
부위원장	허충길
위 원	전병곤
위 원	이재욱
위 원	Derek Dreyer

Abstract

To improve the performance of C programs, mainstream compilers perform aggressive optimizations that may change the behaviors of programs that use low-level features in unidiomatic ways. Unfortunately, despite many years of research and industrial efforts, it has proven very difficult to adequately balance the conflicting criteria for low-level features and compiler optimizations in the design of the C programming language. On the one hand, C should support the common usage patterns of the low-level features in systems programming. On the other hand, C should also support the sophisticated and yet effective optimizations performed by mainstream compilers. None of the existing proposals for C semantics, however, sufficiently support low-level features and compiler optimizations at the same time.

In this dissertation, we resolve the conflict between some of the low-level features crucially used in systems programming and major compiler optimizations. Specifically, we develop the first formal semantics of relaxed-memory concurrency, separate compilation, and cast between integers and pointers that (1) supports their common usage patterns and reasoning principles for programmers, and (2) provably validates major compiler optimizations at the same time. To establish confidence in our formal semantics, we have formalized most of our key results in the Coq theorem prover, which automatically and rigorously checks the validity of the results.

Keywords: C, formal semantics, compiler verification, relaxed-memory concurrency, separate compilation, integer-pointer cast
Student Number: 2013-20737

The outcomes that matter in research are not numerous publications, best-paper awards, completed PhD theses, keynote invitations, software tools, citations and other measurable signs of progress. I was after real success, in the sense of changing the way the IT industry develops software. [...] By that standard, the story told in this article is one of glaring, unremitted and probably definitive failure. —Bertrand Meyer [17]

Acknowledgements

Without great sacrifices of teachers, I would not have been able to finish my Ph.D. Prof. Kwangkeun Yi bore with my childish enthusiasm and tamed it into a dedication. My advisor Prof. Chung-Kil Hur taught me how to read, write, listen, speak, and think by doing together. Dr. Derek Dreyer gave constructive advice that I can follow at just the right times. Now I realize everything they have done to me requires a lot of patience. I would like to say thank you with all my heart.

I would also like to thank Prof. Byung-Gon Chun and Prof. Jae W. Lee for reviewing and giving feedback on this dissertation and my job talk as committee members. I also thank William Mansky, Dmitri Garbuzov, Steve Zdancewic, Viktor Vafeiadis, Yoonseung Kim, and Ori Lahav for working together on the papers which this dissertation is based on, and thank Robbert Krebbers, Jim Apple, Xavier Leroy, Doug Lea, Alan Jeffrey, Andreas Lochbihler, James Riely, Peter Sewell, and Joe Tassarotti for very helpful feedback on the papers.

The papers on which this dissertation based are supported in part by the Engineering Research Center of Excellence Program of MSIP/NRF (Grant NRF-2008-0062609), the ICT R&D program of MSIP/IITP (Grant R0132-15-1006), Samsung Research Funding Center of Samsung Electronics (Project Number SRFC-IT1502-07), a Kwanjeong Educational Foundation Scholarship, and a Korea Foundation for Advanced Studies Scholarship.

I remember my friends. Jaeyeol have always been my teacher, mentor, friend, and brother, and he organized “Kang Sa Mo” for my wedding ceremony. Minsuk and Jonghwan took care of me as if I’m their younger brother. Hyeong Kyun and I went through bad times together as comrades, and Joonhyuk, Taehong, Hyunsung and I went through good times together as comrades. I remember Wonha. I also remember all ROPAS and

SF members, and especially Soonho, Wonchan, and Wontae for helping me in my early career.

None of this could have happened without my family. Mom and Dad, you have always provided me with all that I needed so far. Now I am starting to realize what it meant for you only after having my own child. All I can say is just thank you and love you. Grandma, you colored my childhood with joy and warmth. I miss you so much and I really hope you could attend my commencement ceremony. Minjung, your positive thinking helped us bond together as a family, in good times and in bad. You are probably the greatest sister in the history.

And Eunjung, my lovely and brilliant wife, you sacrificed a lot of things for this dissertation and my career. Thank you for supporting and loving me. This dissertation, as well as the rest of my life, is dedicated to you. Saeun, your smile raises me up even when I am totally exhausted. Someday you will grow up, but I will be your shelter forever. Eunjung and Saeun, I will always love you no matter what.

Contents

Abstract **i**

Acknowledgements **iii**

Chapter I Prologue **1**

1 Introduction 1

1.1 Conflict between Low-Level Features and Compiler Optimizations 4

1.2 Reconciling Low-Level Features with Compiler Optimizations 7

2 Background: A Brief Tour of CompCert 10

2.1 Compiler Correctness 11

2.2 Memory Model 13

2.3 The RTL Language 16

2.4 Constant Propagation 20

Chapter II Relaxed-Memory Concurrency **26**

3 Introduction 26

3.1 Criteria for a Programming Language Memory Model 27

3.2 The "Out of Thin Air" Problem 27

3.3 A "Promising" Semantics for Relaxed Memory 29

4 Basic Model for Handling Relaxed Accesses 30

4.1 Main Ideas 31

4.2 Formal Definition 35

5 Supporting Atomic Updates 37

6 Full Model 40

6.1 Release/Acquire Synchronization 40
6.2 Sequentially Consistent (SC) Fences 44
6.3 “Plain” Non-Synchronizing Accesses 45
6.4 System Calls 46
6.5 Modifying Existing Promises 46
6.6 Formal Model 47
7 Results 51
7.1 Compiler Transformations 51
7.2 Compilation to TSO 53
7.3 DRF Theorems 54
7.4 An Invariant-Based Program Logic 56
8 Proofs 57
8.1 Thread-Local Simulation Relation 57
8.2 Proof of DRF-RA 60
8.3 Proof of DRF-LOCK 64
9 Related Work 66
10 Follow-up and Future Work 69

Chapter III Separate Compilation and Linking 72
11 Introduction 72
12 Overview 75
12.1 Compositional Correctness Level A 75
12.2 Compositional Correctness Level B 77
13 Adapting Constant Propagation to Separate Compilation 80
13.1 Verifying Compositional Correctness Level A 80
13.2 Verifying Compositional Correctness Level B 83
14 Adapting the Other Passes to Separate Compilation 84
14.1 RTL-Level Optimizations that Rely on Value Analysis 85
14.2 Selection 85
14.3 Inlining 87
14.4 SimplExpr 87
15 Results 87
16 Discussion 89
16.1 Related Work 89
16.2 Generality of Our Techniques 92
16.3 Impact 93

Chapter IV Cast between Integers and Pointers **94**

17 Introduction 94

18 Formal Semantics of Hybrid Model 97

18.1 Hybrid of Concrete and Logical Blocks 97

18.2 Combining Logical and Concrete Blocks 97

18.3 Choosing Concrete Blocks 98

18.4 Assigning Concrete Addresses 99

18.5 Operations on Pointers 100

18.6 Dead Cast Elimination 101

18.7 Drawbacks of the Hybrid Model 102

18.8 Language Semantics 103

19 Fix to the LLVM and GCC miscompilation Bugs 105

20 Compiler Verification Techniques and Examples 106

20.1 Specification of Out of Memory 106

20.2 Running Example & Informal Verification 107

20.3 Memory Invariants 109

20.4 Proving Simulation 111

20.5 Examples 112

21 Discussion 114

21.1 Implementation and Experiment Details 114

21.2 Related Work 115

21.3 Compatibility 116

21.4 Impact 118

Chapter V Epilogue **119**

22 Conclusion 119

초록 **129**

List of Figures

Figure I.1 An LLVM bug in the presence of integer-pointer casts 6
Figure I.2 RTL syntax 16
Figure I.3 RTL semantic domains 17
Figure I.4 Operational semantics of RTL 19
Figure I.5 Example of constant propagation 22
Figure I.6 CompCert's simulation relation for the constant propagation . 23

Figure II.1 Operational semantics for only relaxed read and write 34
Figure II.2 Additional rule for updates 40
Figure II.3 Full operational semantics 49
Figure II.4 A simple derivation in the invariant-based program logic . . . 57

Figure III.1 Proving Level A correctness 75
Figure III.2 Proving Level B correctness for RTL passes 78
Figure III.3 A bug due to CompCert 2.4 value analysis 82
Figure III.4 Classification of optimization passes in CompCert 84
Figure III.5 Changes to lines of codes for LevelA and LevelB correctness . 88

Figure IV.1 A GCC bug in the presence of integer-pointer casts 95
Figure IV.2 Arithmetic optimization example I 98
Figure IV.3 Dead code elimination example 99
Figure IV.4 Ownership transfer example 100
Figure IV.5 Arithmetic optimization example II 101
Figure IV.6 Dead cast elimination example 102
Figure IV.7 Memory invariants for the running example 108
Figure IV.8 Memory invariants for hybrid model 110

Chapter I

Prologue

1 Introduction

The C programming language is the *lingua franca* for systems programming, mainly due to its three notable advantages: *low-level features*, *portability*, and *performance*. C provides low-level features that offer programmers precise control over hardware such as pointer manipulation, shared-memory concurrency, and asynchronous interrupt. At the same time, C is portable in that C programs can be compiled and then executed in most of the existing hardware. Furthermore, C provides decent performance: a program written in C are usually outperforming equivalent programs written in other languages—even including carefully hand-written assembly programs—when they perform exactly the same task. These advantages for decades have attracted system programmers, and as a result, a giant ecosystem was built around the C programming language.

C provides low-level features and portability—seemingly conflicting properties—at the same time because it is a balanced abstraction over various hardware assembly languages. If C were exposing too much detail of particular hardware architectures, then it would have not been able to support mismatching architectures, losing a significant degree of portability; on the other hand, if C were exposing too little detail of hardware architectures, then it would have not been able to provide low-level features. The design choice of C as a hardware abstraction is so popular that other systems programming languages—such as C++, D, Objective C, Swift, and Rust—largely follow the design of C and are often called “C-like”.

Compiler Optimizations Being a simple syntax translator from C to assembly languages, however, is insufficient for C compilers to provide the desired level of performance. Instead, compilers perform *optimizations* that transform the given program to be executed more efficiently in hardware while preserving its semantics. Optimizations are so effective that they have been an essential ingredient of compilers since the early days. For example, every system programmer expects a compiler to perform quite sophisticated optimizations such as register promotion [5] and register allocation [6]. Optimizations are becoming more important these days because recent hardware trends—such as SIMD, GPU, and accelerators—offer potential for compiler optimizations to further improve the performance of systems.

The following is an instance of the *constant propagation* optimization, which significantly improves the performance of compilation results and is thus performed by mainstream compilers such as GCC [2] and LLVM [3]:

```
void f() {                void f() {
1: int a = 42;            1: int a = 42;
2: g();             ⇝     2: g();
3: print(a);              3: print(42); // const. prop.
}                         }
```

Suppose `g()` is an external function whose body is unknown to the compiler, and `print(a)` prints the value of `a` to the screen. The function `f()` first assigns `42` to the local variable `a` (line `1`), calls some unknown external function `g()` (line `2`), and then prints `a` (line `3`). As an optimization, mainstream compilers replace `a` with `42` at line `3`, effectively propagating the constant `42` at line `1` to line `3`. Compilers perform such an optimization even in the presence of a function call to the unknown function `g()`, because—at least in the viewpoint of compilers—the address of the variable `a` is not leaked to `g()` and thus its content cannot be modified by `g()`.

This optimization, however, may change the program’s behavior, putting the soundness of the optimization in danger. For instance, suppose that `f()` is linked with the following `g()`:

```
void g() {
1: int anchor;
2: int *guess = &anchor + 10; // guessing &a
3: *guess = 666;
}
```

Here, the function `g()` tries to *guess* the address of `a` by exploiting the fact that stack usually grows downwards with a fixed offset: it first declares a variable `anchor` and guesses that `a` is located 10 words later than than `anchor` is. While extremely dangerous and thus discouraged, the guess sometimes happens to be correct, invalidating the compiler's reasoning that `a` is accessible only within the function `f()`. If it is the case, when linked with `g()`, the original `f()` will print the evil value 666; on the other hand, the optimized `f()` will print the expected value 42.[1]

To rescue the soundness of constant propagation, the ISO C18 standard [37] blames `g()` for violating the rule of the C programming language by marking it as invoking *undefined behavior* [37, §3.4.3p1]. Specifically, line `2` invokes undefined behavior because `guess` is derived from `anchor` and yet it does not point to the valid location within `anchor`'s allocation [37, §6.5.6p8]. (Roughly speaking, all the pointers derived from `anchor` *shall* point to `anchor`; otherwise, the behavior is undefined.) Now an instance of undefined behavior allows compilers to do anything it chooses, from arbitrarily changing the code and thereby justifying the constant propagation optimization to even making "demons fly out of your nose" [8].[2]

Notice that C intentionally loses the ability to manipulate pointers in an unrestricted way—*e.g.*, deriving the address of `a` from that of `anchor`—in order to justify the constant propagation optimization. This is beneficial because the performance improvement offered by compiler optimizations outweights the cost of the restriction on the low-level feature for unidiomatic programs. As a result, pointers in C should have a richer structure than those in assembly languages that have the same representation as integer values of the appropriate width and simply index into a single flat array representing memory.

[1] We got these results by separately compiling `f()` and `g()` and then linking them using GCC 8.2.1 with compile option `-fno-stack-protector` in an x86-64 machine running Linux 4.20.

[2] Notice that undefined behavior is not necessary for higher-level languages—such as Java, C#, OCaml, Haskell—to justify compiler optimizations thanks to their lack of low-level features. For example, constant propagation is immediately justified in Haskell without resorting to undefined behavior thanks to its lack of raw pointer.

Such a practice of taming low-level features for supporting compiler optimizations is actually quite common in the design of the C programming language: mainstream compilers perform aggressive optimizations that may change the behaviors of programs that use low-level features in unidiomatic ways. As a result, C programs may have different meaning than the exactly same programs written in assembly languages (modulo syntactic differences). In other words, C is no longer just a thin wrapper around assembly languages but it should rather be an abstraction over both assembly languages *and* compiler optimizations.

1.1 Conflict between Low-Level Features and Compiler Optimizations

Unfortunately, despite many years of academic and industrial efforts [37], it has proven very difficult to adequately balance the conflicting criteria for low-level features and compiler optimizations in the design of the C programming language. On the one hand, C should support the common usage patterns of the low-level features in systems programming, such as relaxed-memory concurrency, separate compilation, and cast between integers and pointers. In addition, programmers should be able to reason about programs that use the low-level features. On the other hand, C should also support the sophisticated and yet effective optimizations performed by mainstream compilers, such as register promotion, constant propagation, and dead code elimination. To the best of our knowledge, none of the existing proposals for C semantics sufficiently support both low-level features and compiler optimizations at the same time.

Prior Work The ISO C standard, even after a series of revisions including C89, C99, C11, and C18, still has quite unclear specification for some of the low-level features as of this writing. First, ISO C18 [37] informally describes the C programming language in English prose, which is often ambiguous and confusing. The problem has only worsened by the fact that the description contains many ad-hoc exceptions including 203 cases of undefined behavior [37, J.2]. Second, ISO C18 intentionally leaves the precise meaning of some of the low-level features undefined. For example, the semantics of cast between integers and pointers is not properly defined in ISO C18, while it is essential for applications such as operating system kernels and language runtimes.

Accordingly, there have been numerous efforts to capture the subtleties of the ISO C standard by giving an alternative formal language definition [56, 83, 26, 63, 46]. However, all these projects—while supporting a significant subset of ISO C99 or LLVM IR—make unrealistic simplifying assumptions on C semantics and lack support for various low-level features.

As a result, many systems programming communities, *e.g.*, the Linux kernel developers, use their own *dialect* of C that is closer to assembly languages and supports more low-level features and less compiler optimizations than the standard. The dialects, however, are often informally described as the set of turned-on compiler optimizations (*e.g.*, "`gcc -O2`"), whose meaning is unclear and unstable.

Problem The unresolved conflict between low-level features and compiler optimizations causes difficulties to both programmers and compiler writers. For programmers, it is difficult to expect how programs that use low-level features will behave because compilers may perform conflicting optimizations, which introduce surprising non-local changes and difficult-to-find bugs in program behavior [80, 81]. As a result, mainstream compilers are typically unused for safety-critical systems or used with only few compiler optimizations turned on, significantly increasing verification cost and degrading performance of safety-critical systems. One the other hand, for compiler writers, it is difficult to figure out whether an optimization is sound or not in the presence of low-level features. Even worse, sometimes a combination of optimizations—while each and every one of them seems legit—results in miscompilation bugs, for which it is unclear how to fix.

A Miscompilation Bug Figure I.1 presents an LLVM miscompilation bug[3] due to conflicting optimizations. Note that the type `uintptr_t` is an integer type that is capable of holding a pointer value [37, §7.20.1.4]. For this program, the expected outcome is either `a=0 x=15` or `a=100 x=0` for the following reasons:

- Suppose `n=0`. Then `pi` points to `x` after line `12`. Thus line `13` writes `15` to `x`, and the end result is `a=0 x=15`.
- Suppose otherwise. Then `a=100` and `pi` points to `y` after line `12`. Thus line `13` writes `15` to `y`, and the end result is `a=100 x=0`.

However, the result `a=0 x=0` is observed when `c.c` and `b.c` are compiled with `clang -O2` and then executed due to the following series of optimizations:

1. The integer comparison `pi != yi` at line `4` is replaced with the pointer comparison `&x != y+1`.

[3] This bug is reported in the LLVM bug tracker: `https://bugs.llvm.org/show_bug.cgi?id=34548`

```
// c.c
#include <stdio.h>
#include <stdint.h>

void f(int*, int*);

int main()
{
 1: int a=0, y[1], x = 0;
 2: uintptr_t pi = (uintptr_t) &x;
 3: uintptr_t yi = (uintptr_t) (y+1);
 4: uintptr_t n = pi != yi;

 5: if (n) {
 6:   a = 100;
 7:   pi = yi;
 8: }
 9: if (n) {
10:   a = 100;
11:   pi = (uintptr_t) y;
12: }

13: *(int *)pi = 15;

14: printf("a=%d x=%d\n", a, x); // observed: a=0 x=0
15: f(&x,y);
16: return 0;
}

// b.c
void f(int*x, int*y) {}
```

Figure I.1 An LLVM bug in the presence of integer-pointer casts

2. The compiler assumes `n=0`, which is allowed since now line `4` is comparing pointers from different origins.

3. Lines `5-12` is eliminated since the condition `n` evaluates to false.

4. `(int*)pi` at line `13` is replaced with `(int*)yi` since `n=0`, and then with `y+1` since `yi=(uintptr_t)(y+1)`.

5. Line `13` is eliminated since it is writing to an invalid address `y+1`. Then line `14` prints `a=0 x=0`.

In short, LLVM has a miscompilation bug due to the conflict among the above five optimizations. However, each of them looks legit—at least in the viewpoint of the LLVM compiler—and it is unclear which one(s) is to blame. That is one of the reasons this bug is still open in the bug tracker as of this writing.

1.2 Reconciling Low-Level Features with Compiler Optimizations

In this dissertation, we resolve the conflict between some of the low-level features crucially used in systems programming and major compiler optimizations. Specifically, we **develop formal semantics** of **relaxed-memory concurrency** (Chapter II), **separate compilation** (Chapter III), and **cast between integers and pointers** (Chapter IV) that (1) supports their common usage patterns and reasoning principles, and (2) provably validates major compiler optimizations at the same time.

Our formal semantics is beneficial to both programmers and compiler writers. Since formal semantics is a mathematically clear definition of program behaviors, it makes possible for programmers to expect how compiled programs will behave regardless of which optimizations are performed by compilers. On the other hand, compiler writers can figure out whether an optimization is sound or not using formal semantics as the criteria. In particular, with formal semantics, we can point out which optimization(s) is to blame in the miscompilation bug above.

To establish confidence in our formal semantics, we prove the soundness of compiler optimizations in the presence of the low-level features. The soundness proof guarantees that they preserve the semantics of source programs and do not introduce any bugs. The absence of miscompilation bugs ensures higher level of reliability and thus enables optimizations to be used even for safety-critical systems with confidence. **We have formalized all the soundness proofs reported in this dissertation in the Coq theorem prover** [35], which automatically and rigorously checks the validity of the soundness proofs. The formalization is available online [1].

In the rest of this section, we will briefly describe our main contributions, namely developing formal semantics of three low-level features of C.

Chapter II: Relaxed-Memory Concurrency Relaxed-memory concurrency is the study of *relaxed behaviors*, which are observable behaviors of concurrent programs beyond those allowed in *sequential consistency* (think: interleaving of the executions of threads). Relaxed behaviors are made possible due to hardware and compiler optimizations such as out-of-order execution or instruction reordering/merging. While relaxed behaviors complicate reasoning of concurrent programs, they are unavoidable in effectively exploiting the parallelism provided by shared-memory architecture. Despite many years of research, however, it has proven very difficult to develop a formal semantics for programming languages with relaxed-memory concurrency that adequately balances the conflicting desiderata of programmers and compilers.

In this chapter, we propose the first formal semantics of relaxed-memory concurrency that (1) justifies simple invariant-based reasoning, thus demonstrating the absence of bad "out-of-thin-air" behaviors, (2) supports "DRF" guarantees, ensuring that programmers who use sufficient synchronization need not understand the full complexities of relaxed-memory semantics, (3) is implementable, in the sense that it provably validates many standard compiler optimizations and reorderings, as well as standard compilation schemes to x86-TSO, (4) accounts for a broad spectrum of low-level concurrency features in C, and (5) defines the semantics of racy programs without relying on undefined behaviors, which is a prerequisite for applicability to type-safe languages like Java. The key novel idea behind our semantics is the notion of promises: a thread may make the effect of a write before actually executing it, thus enabling other threads to read from that write out of order. Crucially, to prevent out-of-thin-air behaviors, a promise step requires a thread-local certification that it will be possible to execute the promised write even in the absence of the promise.

Our semantics draws interest from both industry and academia. Our semantics not only serves as a guide to C/C++ relaxed-memory concurrency [7, 4] but also influences the discussion on the standard semantics for C/C++ and compiler IRs (private communication in mailing lists). Svendsen *et. al.* developed reasoning principles for our semantics [77], and Podkopaev *et. al.* validated compilation schemes for our semantics to various architectures such as ARMv7, ARMv8, RISC-V, and Power [68]. Furthermore, the idea of promises and certification is also used to model the relaxed-memory concurrency in ARMv8 and RISC-V [70].

This chapter draws heavily on the work and writing in the following paper:

[40] **Jeehoon Kang**, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, Derek Dreyer. *A Promising Semantics for Relaxed-Memory Concurrency*. **POPL 2017**.

Chapter III: Separate Compilation and Linking Separate compilation and linking is essential in practice because they significantly reduces compilation time. However, major compiler verification efforts, such as CompCert and Vellvm, have traditionally simplified the verification problem by restricting attention to the correctness of whole-program compilation, leaving open the question of how to verify the correctness of separate compilation. Recently, a number of sophisticated techniques have been proposed for proving more flexible, compositional notions of compiler correctness, but these approaches tend to be quite heavyweight compared to the simple "closed simulations" used in verifying whole-program compilation. Applying such techniques to a compiler like CompCert, as Stewart *et. al.* have done [76], involves major changes and extensions to its original verification.

In this chapter, we show that if we aim somewhat lower—to prove correctness of separate compilation, but only for a single compiler—we can drastically simplify the proof effort. Toward this end, we develop several lightweight techniques that recast the compositional verification problem in terms of whole-program compilation, as far as the compiler's transformations and optimizations satisfy what we call *monotonicity*. The proof techniques enable us to largely reuse the closed-simulation proofs from existing compiler verifications.

We demonstrate the effectiveness of these techniques by applying them to CompCert 2.4, converting its verification of whole-program compilation into a verification of separate compilation in less than two person-months. This conversion only required a small number of changes to the original proofs. Along the way, we uncovered two compiler bugs—one of which is on separate compilation and the other is orthogonal to separate compilation—and our proof techniques are subsequently adopted in CompCert 2.7.

This chapter draws heavily on the work and writing in the following paper:
[42] **Jeehoon Kang**, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, Viktor Vafeiadis. *Lightweight Verification of Separate Compilation*. **POPL 2016**.

Chapter IV: Cast between Integers and Pointers Cast between integers and pointers is one of the defining characteristic of the C programming language in that it allow cross-platform low-level manipulation of memory layout, which is essential for applications such as operating system kernels and language runtimes. However, the feature drastically conflicts with major compiler optimizations, as demonstrated by the example shown in Figure I.1. The ISO C standards try to reconcile the feature and the optimizations using the notion of *provenance*, but it fails to support certain common optimizations and requires an intrusive change to the language semantics.

In this chapter, we propose the first formal semantics of cast between integers and pointers that (1) fully supports operations on the representation of pointers, including all arithmetic operations for pointers that have been cast to integers, (2) validates major compiler optimizations on memory accesses, and (3) is simple to understand and program with. The key novel idea behind our semantics is the notion of *concretization*: when allocated, a memory block is not assigned a concrete address yet; only when it is required by a pointer-to-integer cast, the block is lazily assigned a concrete address, *i.e.*, the block is concretized.

Along the way, we discovered a GCC bug in the presence of integer-pointer casts, which clearly shows it is safe to turn off too aggressive alias analyses. Furthermore, our idea has subsequently been refined by follow-up papers by other researchers [54, 60], which are accompanied with promising revision proposals to the LLVM IR and the ISO C standard.

This chapter draws heavily on the work and writing in the following paper:
[41] **Jeehoon Kang**, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, Viktor Vafeiadis. *A Formal C Memory Model Supporting Integer-Pointer Casts*. **PLDI 2015**.

Organization The rest of this chapter provides the technical background on formal semantics and compiler verification that informs the rest of this dissertation (Section 2). Chapters II to IV present the main contributions of this dissertation. This dissertation concludes with Chapter V, which summarizes its contributions, impacts, and future work (Section 22).

2 Background: A Brief Tour of CompCert

Before delving into the main contributions, we briefly review the technical background on formal semantics and compiler verification that informs the rest of this dissertation, using the CompCert C compiler [55] as the learning material. The technical ideas and results presented in this section apply not only to CompCert but also to other formal semantics and compiler verification projects as well.

The CompCert project was initiated by Xavier Leroy over ten years ago and then grows as the first realistic verified compiler. CompCert is verified in the sense that it "is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program." As such, CompCert guarantees that program analyses and verifications performed on its input carry over soundly to its machine-level output. CompCert is realistic in the sense that it "could realistically be used in the context of production of critical software". It compiles a significant subset of ISO C99 down to several architectures, and it performs a number of common and useful optimizations. It received significant interest from the avionics industry [14, 75], and recently, it is certified for being used for nuclear power plant [50]. It has also served as a fundamental building block in academic work on end-to-end verified software [12].

In this section, we first explain CompCert's correctness statement, as well as its simulation verification technique (Section 2.1). To flesh out the details on formal semantics and compiler verification, we use RTL—one of CompCert's internal representations—and constant propagation—one of CompCert's optimizations—as a running example. Specifically, We first review CompCert's memory model that is specifically designed to verify compiler optimizations (Section 2.2). Then we explain the RTL language on which constant propagation and most of the other optimizations are performed in the CompCert's compilation pipeline (Section 2.3), and how constant propagation works and how CompCert verifies it (Section 2.4). Throughout this section, we keep the presentation semi-formal, abstracting away unnecessary detail to get across the main ideas. For more details, we refer the reader to [55, 57].

2.1 Compiler Correctness

End-to-End Correctness Roughly speaking, the correctness result of CompCert can be understood to assert *semantic preservation*, which in turn means the following. Suppose `s.c` is a "source" file (in C), `t.asm` is a "target" file (in assembly), and $\mathcal{C}$ is a verified compiler (represented as a function from C files to assembly files).

$$\frac{\mathcal{C}(\texttt{s.c}) = \texttt{t.asm} \qquad s = \mathrm{load}(\texttt{s.c}) \qquad t = \mathrm{load}(\texttt{t.asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

If `t.asm` is the result of compiling `s.c` with $\mathcal{C}$, then executing `t.asm` according to assembly semantics will result in a subset of the behaviors one could observe from executing `s.c` according to C semantics. (We write $s = \mathrm{load}(\texttt{s.c})$ to denote the *machine state* that results from loading `s.c` into memory, $\mathrm{Behav}(s)$ to denote the observable behaviors of the execution of s, and analogously for t and `t.asm`.) Hence, we say that `t.asm`, the target-level output of $\mathcal{C}$, *refines* its source-level input, `s.c`.

Notice that the compiler correctness statement presented here will be generalized to support separate compilation in Chapter III.

Set of Behaviors We consider sets of behaviors as opposed to single behaviors because a program may produce multiple behaviors due to nondeterminism. Given a set of I/O events that programs may generate and users may observe, a behavior is one of the following three forms: (1) a terminating execution producing a finite sequence of I/O events, $e_1, \cdots, e_n, \texttt{term}$; (2) a diverging execution that has produced only a finite sequence of I/O events, $e_1, \cdots, e_n, \texttt{nonterm}$; and (3) a diverging execution producing an infinite sequence of I/O events, $e_1, \cdots, e_n, \cdots$.

Undefined behavior requires special attention in defining the set of behaviors, because the program states in the condition are neither terminated nor transitioning to other states, fitting into none of the behavioral categories. We assign *the set of all behaviors* to the program states invoking undefined behavior in order to validate compiler optimizations: if the source program's behavior is undefined, then compiler can choose any program as its result; on the other hand, if the target program's behavior is undefined, then the source program's behavior should also be undefined.

Notice that the notion of behaviors presented here will be generalized to support out-of-memory in Chapter IV.

Per-Pass Correctness To verify compilation correctness for the compiler $\mathcal{C}$, CompCert verifies each pass of $\mathcal{C}$ independently. Specifically, for each pass (transformation) $\mathcal{T}$ from language L_1 to L_2—where the L_i's may be C, assembly, or some intermediate languages—we show the following:

$$\frac{\mathcal{T}(\texttt{s.l1}) = \texttt{t.l2} \qquad s = \text{load}(\texttt{s.l1}) \qquad t = \text{load}(\texttt{t.l2})}{\text{Behav}(s) \supseteq \text{Behav}(t)}$$

That is, given the input `s.l1` and output `t.l2` of the $\mathcal{T}$ transformation, we show that the behaviors of `t.l2` are contained within those of `s.l1`. Since subset inclusion is transitive, it easy to see that the proofs of the constituent passes of $\mathcal{C}$ compose to establish the correctness of $\mathcal{C}$ as a whole.

Verifying Per-Pass Correctness Now how does one actually prove the verification condition for each individual pass? The standard approach taken by CompCert is to use (closed) simulations. Informally, we will say that a *simulation* R is a relation between running programs (*i.e.*, machine states) in L_1 and L_2 such that, if $(s, t) \in R$, then the behaviors one observes while stepping through the execution of t are matched by corresponding observable behaviors in the execution of s. One can think of R as imposing an invariant, which describes (and connects) the possible machine states of the source and target programs, and which must be maintained as the programs execute.

We leave further details about simulations until Section 2.4; suffice it to say that they satisfy the following "adequacy" property:

$$\frac{R \text{ is a simulation} \qquad (s, t) \in R}{\text{Behav}(s) \supseteq \text{Behav}(t)}$$

Thus, to establish the verification condition for pass $\mathcal{T}$, it suffices to exhibit a simulation R that relates load(`s.l1`) and load(`t.l2`).

In the rest of this section, we flesh out the details on formal semantics and compiler verification in CompCert, using constant propagation as the running example. Constant propagation is essentially a transformation that optimizes memory operations, so we will first review their semantics in CompCert.

2.2 Memory Model

CompCert supports the following memory operations:

$$\begin{aligned}
\mathit{load} &\in \text{Mem} \to \text{Addr} \to \mathbb{P}(\text{Val}) \\
\mathit{store} &\in \text{Mem} \to \text{Addr} \to \text{Val} \to \mathbb{P}(\text{Mem}) \\
\mathit{malloc} &\in \text{Mem} \to \texttt{uint32} \to \mathbb{P}(\text{Addr} \times \text{Mem}) \\
\mathit{free} &\in \text{Mem} \to \text{Addr} \to \mathbb{P}(\text{Mem})
\end{aligned}$$

Memory, $m \in \text{Mem}$, supports four operations: load, store, malloc, and free. Note that all operations may produce multiple results due to the nondeterminism arising from *e.g.*, concurrency. A load operation reads the value, $v \in \text{Val}$, of an address, $a \in \text{Addr}$, and a store operation writes a value to an address in the memory. If a load or store operation access an illegal address, then there are no valid results. A malloc operation allocates a memory object of the specified size of type `uint32` (the set of 32-bit unsigned integers), returning the new memory and the address of the allocated object, and a free operation deallocates an allocated memory object.

Now we define the semantic domain for and the semantics of those memory operations, or in other words, define the *memory model*. The memory model presented here will be generalized to support integer-pointer casts and relaxed-memory concurrency in Chapter IV and Chapter II, respectively.

To simplify the presentation, we do not discuss many aspects of C memory models that are orthogonal to the contributions of this dissertation. Specifically, we (1) assume a 32-bit architecture: words are 4 bytes wide and the size of the address space is 2^{32}, as eminent from the signature of *malloc*; (2) consider only unsigned integer and pointer values, and omit values of other types such as `int32`, `float` or `char`; and (3) omit subword arithmetic, and assume each address stores a 32-bit value.

Concrete Model The most straightforward way to define a memory model is closely following the assembly language: pointers have the same representation as integer values of the appropriate width, and they simply index into a single flat array representing memory. In such a *concrete memory model*, memory consists of a 2^{32}-sized array of values, and a list of allocated blocks, represented as pairs (p, n) of the block's starting address and its size. Loading from or storing to an unallocated address raises an error (*i.e.*, undefined behavior). Values are just 32-bit integers, since pointers are merely integers in the concrete model.

$$\begin{aligned}
\mathrm{Mem} &\overset{\mathrm{def}}{=} (\texttt{uint32} \rightarrow \mathrm{Val}) \times \texttt{list}\ \mathrm{Alloc} \\
\mathrm{Alloc} &\overset{\mathrm{def}}{=} \{\, (p, n) \mid p \in \texttt{uint32} \land n \in \texttt{uint32} \,\} \\
\mathrm{Val} &\overset{\mathrm{def}}{=} \{\, i \in \texttt{uint32} \,\}
\end{aligned}$$

Memory allocation inserts a block into the list of allocated blocks, whereas deallocation removes one. Overall, the list of allocated blocks should be *consistent*:[4]

- If (p, n) is allocated, then $\varnothing \neq [p, p + n) \subseteq (0, 2^{32} - 1)$.
- If blocks (p_1, n_1) and (p_2, n_2) are distinct allocations, their ranges $[p_1, p_1 + n_1)$ and $[p_2, p_2 + n_2)$ are disjoint.

The concrete model, however, *does not support standard compiler optimizations*. As we have seen in Section 3, constant propagation is unsound on the concrete model in the presence of external function calls, because the model does not provide a mechanism for ensuring that a module has exclusive control over some part of memory, and as a result, programmers should pessimistically assume that unknown code can read and update the contents of every allocated memory cell. Furthermore, dead allocation elimination is also unsound in the concrete model. For example, the following transformation—removing the unused local variable `a`—might change the behavior of the program: by virtue of there being one fewer memory cells allocated, the call to `g()` might succeed

[4] These are a subset of `malloc`'s properties according to the ISO C18 standard. For more details, see [37, §7.22.3p1 and §6.5.8p5].

where initially it exhausted memory:

```
void f() {                void f() {
int a = 42;               // dead allocation elimination
g();                  ⇝   g();
print(42);                print(42);
}                         }
```

CompCert's Logical Model In order to support standard compiler optimizations, CompCert is based on a *logical* memory model [56, 57] rather than the naive concrete model. In the logical model, memory is a finite collection of logical blocks with unique block identifiers, together with the maximum of the allocated block identifiers. Each block is a contiguous, fixed-sized array of values annotated with a validity flag v that indicates whether the block is accessible or has been freed. As before, accessing a freed block raises an error. Values are either 32-bit integers, logical addresses, or the special *undefined value* that represents uninitialized data. Here, a logical address (l, i) consists of a block identifier l and an offset i inside the block.

$$\begin{aligned}
\text{Mem} &\overset{\text{def}}{=} \{\, (nb, bs) \mid nb \in \text{BlockId} \land bs \in \text{BlockId} \rightharpoonup_{\text{fin}} \text{Block} \,\} \\
\text{Block} &\overset{\text{def}}{=} \{\, (v, n, c) \mid v \in \{\texttt{valid}, \texttt{freed}\} \land n \in \mathbb{N} \land c \in \text{Val}^n \,\} \\
\text{Val} &\overset{\text{def}}{=} \{\, i \in \texttt{uint32} \,\} \uplus \text{Addr} \uplus \{\texttt{undef}\} \\
\text{Addr} &\overset{\text{def}}{=} \{\, (l, i) \in \text{BlockId} \times \texttt{uint32} \,\}
\end{aligned}$$

An important advantage of the logical model over the concrete one is that it *allows functions to have exclusive control over a logical block* as long as they do not allow its address to escape. The reason is that it is not possible to manufacture the logical address of an already allocated block. This property guarantees the correctness of many useful optimizations, such as constant propagation across function calls and dead allocation elimination. A secondary advantage is that programs have *infinite memory*, rendering their allocation behavior unobservable, which in turn makes it easy for compilers to remove dead allocations.

Apart from that logical models have a slightly more complicated semantics, their main disadvantage is that *they do not support integer-to-pointer casts* very well. As a result, CompCert does not support real-world use cases of integer-pointer casts well. Specifically, they are treated as `nop`s (*i.e.*, the identity function), and thus variables of integer (or pointer) types can contain both integers and logical addresses. This conflicts with the intention of integer-pointer casts to freely manipulated pointers as if they are integer values. We will address this disadvantage in Chapter IV.

$$
\begin{array}{lcll}
\text{Prog} & ::= & \overline{\text{Decl}} & \\
\text{Decl} & ::= & \texttt{extern}\ [\texttt{const}]\ \text{Id}[\texttt{uint32}] & \text{// External Variable} \\
 & \mid & [\texttt{const}]\ \text{Id}[\texttt{uint32}] := \{\ \overline{\text{GVal}}\ \} & \text{// Variable} \\
 & \mid & \texttt{extern}\ \text{Id FSig} & \text{// External Function} \\
 & \mid & \text{Id FDef} & \text{// Function} \\
\text{FSig} & ::= & (\overline{\text{Reg}}) & \text{// Function Signature} \\
\text{FDef} & ::= & (\overline{\text{Reg}})\ \{\ \overline{\text{Reg};}\ \texttt{sp}[\texttt{uint32}]\ ;\ \text{Code}\ \} & \text{// Function Definition} \\
\text{Code} & ::= & \overline{\text{NodeId : Instr}} & \\
\text{Instr} & ::= & \text{Reg} := \text{FVal} \qquad \texttt{jmp}\ \text{NodeId} & \text{// Immediate Value} \\
 & \mid & \text{Reg} := \mathit{op}\ \overline{\text{Reg}} \qquad \texttt{jmp}\ \text{NodeId} & \text{// Operation} \\
 & \mid & \text{Reg} := \text{FVal}[\texttt{uint32}]\ \ \texttt{jmp}\ \text{NodeId} & \text{// Load} \\
 & \mid & \text{FVal}[\texttt{uint32}] := \text{Reg}\ \ \texttt{jmp}\ \text{NodeId} & \text{// Store} \\
 & \mid & \mathit{cond\text{-}op}\ \overline{\text{Reg}}\ ?\ \texttt{jmp}\ \text{NodeId} : \texttt{jmp}\ \text{NodeId} & \text{// Conditional} \\
 & \mid & \text{Reg} := \text{FVal}(\overline{\text{Reg}}) \qquad \texttt{jmp}\ \text{NodeId} & \text{// Call} \\
 & \mid & \texttt{return}\ \text{Reg} & \text{// Return} \\
 & \mid & \ldots & \\
\text{GVal} & ::= & \text{Id} \mid \texttt{uint32} \mid \texttt{undef} & \\
\text{FVal} & ::= & \text{GVal} \mid \text{Reg} \mid \texttt{sp} & \\
\texttt{uint32} & ::= & \text{the set of 32-bit unsigned integers} & \\
\text{Id} & ::= & \text{the set of identifiers for variables and functions} & \\
\text{Reg} & ::= & \text{the set of register names} & \\
\text{NodeId} & ::= & \text{the set of node labels} &
\end{array}
$$

Figure I.2 RTL syntax

It is worth noting that the logical model is not intended to replace the memory model in the ISO C18 standard. It is a formal refinement of the (informal) standard that can be used for formally reasoning about programs and program transformations (as in compiler verification).

2.3 The RTL Language

So far we discussed CompCert's logical memory model and its advantages. To provide a more concrete context for formal semantics and compiler verification, we explain the syntax and semantics of CompCert's register transfer language (RTL), the compiler's internal language where most of its optimizations take place. For presentation purposes, we simplify the language a bit by removing types and other unnecessary details.

$$
\begin{aligned}
\mathrm{GEnv} &\overset{\mathrm{def}}{=} \{\, (g, d) \mid g \in \mathrm{Id} \rightharpoonup_{\mathrm{fin}} \mathrm{BlockId} \wedge \\
&\qquad\qquad d \in \mathrm{BlockId} \rightharpoonup_{\mathrm{fin}} \mathrm{FSig} \uplus \mathrm{FDef} \,\} \\
\mathrm{State} &\overset{\mathrm{def}}{=} \mathrm{IState} \uplus \mathrm{CState} \uplus \mathrm{RState} \\
\mathrm{IState} &\overset{\mathrm{def}}{=} \{\, \mathtt{ist}\ m\ s\ fd\ sp\ pc\ rs \mid \\
&\quad m \in \mathrm{Mem} \wedge s \in \overline{\mathrm{StkFrm}} \wedge fd \in \mathrm{FDef} \wedge \\
&\quad sp \in \mathrm{Addr} \wedge pc \in \mathrm{NodeId} \wedge rs \in \mathrm{Reg} \rightharpoonup_{\mathrm{fin}} \mathrm{Val} \,\} \\
\mathrm{CState} &\overset{\mathrm{def}}{=} \{\, \mathtt{cst}\ m\ s\ fds\ vs \mid m \in \mathrm{Mem} \wedge s \in \overline{\mathrm{StkFrm}} \wedge \\
&\quad fds \in \mathrm{FDef} \uplus \mathrm{FSig} \wedge vs \in \overline{\mathrm{Val}} \,\} \\
\mathrm{RState} &\overset{\mathrm{def}}{=} \{\, \mathtt{rst}\ m\ s\ v \mid m \in \mathrm{Mem} \wedge s \in \overline{\mathrm{StkFrm}} \wedge v \in \mathrm{Val} \,\} \\
\mathrm{StkFrm} &\overset{\mathrm{def}}{=} \{\, (r, fd, sp, pc, rs) \mid r \in \mathrm{Reg} \wedge fd \in \mathrm{FDef} \wedge \\
&\quad sp \in \mathrm{Addr} \wedge pc \in \mathrm{NodeId} \wedge rs \in \mathrm{Reg} \rightharpoonup_{\mathrm{fin}} \mathrm{Val} \,\}
\end{aligned}
$$

Figure I.3 RTL semantic domains

Syntax The syntax of the CompCert's RTL is given in Figure I.2. Programs are just a list of global declarations, which consist of (1) declarations of external variables and functions provided by different compilation units, and (2) definitions of variables and functions provided by the current compilation unit. For global variable declarations and definitions, we also specify a (non-negative) integer number denoting the size of the declared block in bytes.

Function declarations only contain the function signature, which is a list of parameters, but function definitions additionally contain a list of local registers, the size of their stack frame, and the code. The code is essentially a control-flow graph of three-address code: it is represented as a mapping from node identifiers to instructions, where instructions either do some local computation (*e.g.*, write a constant to a register, or perform some arithmetic computation), load from a memory address, store to memory, do a comparison, call a function, or exit the function and return a result. Each instruction also stores the node identifier(s) of its successor instruction(s).

Throughout we assume that programs satisfy some basic well-formedness properties: there cannot be multiple definitions for the same global variable, declarations and definitions of the same variable should have matching signatures, and the parameter and local variable lists for each function do not have duplicate entries.

Semantics The semantic domain used in CompCert's RTL is given in Figure I.3.

A global environment, $ge = (g, d) \in \mathrm{GEnv}$, maps each global variable name to a logical block identifier, and each logical block identifier corresponding to some function's code to either the corresponding function signature for external functions or the corresponding function definition for functions defined in the program.

Program states can be of three kinds: normal instruction states (`ist`), call states (`cst`) just before passing control to an invoked function, and return states (`rst`), just after returning from an invoked function. Instruction states store the memory (m), the sequence of parent stack frames (s), the definition of the function whose body is currently executed (fd), the current stack pointer (sp), the program counter (pc), and the contents of the local registers (rs). Call states record the memory, the stack, and the function to be called (fds) with its arguments ($args$). The function to be called can be either an internal function, in which case we record its definition, or an external one, in which case we record its signature. Return states record just the memory, the stack, and the value that was returned by the function. A stack s is a list of stack frames, each of which records the same information as normal instruction states, except with the addition of a register name r where its return value should be stored, and minus the memory (m) and stack (s) components.

The meaning of programs is described by three definitions:

$$\begin{aligned} \textit{get-genv} &\in \text{Prog} \to \text{GEnv} \\ \textit{load} &\in \text{Prog} \rightharpoonup \text{State} \\ \hookrightarrow &\in \mathbb{P}(\text{GEnv} \times \text{State} \times \text{Event} \times \text{State}) \end{aligned}$$

The first function, $\textit{get-genv}(prg)$, returns the global environment corresponding to the program: it "allocates" the global variables of the program sequentially in blocks 1, 2, 3, and so on, and maps the blocks corresponding to function symbols to the relevant function definition or signature. Similarly, $\textit{load}(prg)$ returns the initial state obtained by loading a program into memory: it initializes the memory m with the initial values of the global variables at the appropriate addresses generated by $\textit{get-genv}(prg)$, and returns a call state, `cst` m [ ] fd [ ], where fd is the function definition corresponding to `main()`. Loading is a partial function because it is undefined for programs without a `main()` function.

The $\hookrightarrow$ relation is a small-step reduction relation describing how program states evolve during the computation. For clarity, we write $s \overset{\sigma}{\hookrightarrow}_{ge} s'$ instead of $(ge, s, \sigma, s') \in \hookrightarrow$. The operational semantics for RTL is fairly standard and shown in Figure I.4: there is a rule for each of the various basic instructions of the language. Starting from normal instruction states, the instruction at the node pointed to by the program counter is scrutinized ($fd@pc$). Depending on what instruction is there, only one rule is applicable.

(IMM)

$$\frac{\begin{array}{c} fd@pc = (dst := src\ \texttt{jmp}\ pc') \\ rs' = rs[dst \leftarrow [\![src]\!]\,(ge, sp, rs)] \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

(OP)

$$\frac{\begin{array}{c} fd@pc = (dst := op\ args\ \texttt{jmp}\ pc') \\ rs' = rs[dst \leftarrow [\![op]\!]\,(ge, sp, [\![args]\!]\,(rs))] \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

(LOAD)

$$\frac{\begin{array}{c} fd@pc = (dst := src[n]\ \texttt{jmp}\ pc') \\ (l, i) = [\![src]\!]\,(ge, sp, rs) \\ rs' = rs[dst \leftarrow m[(l, i + n)]] \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs'}$$

(STORE)

$$\frac{\begin{array}{c} fd@pc = (dst[n] := src\ \texttt{jmp}\ pc') \\ (l, i) = [\![dst]\!]\,(ge, sp, rs) \\ m' = m[(l, i + n) \leftarrow [\![src]\!]\,(rs)] \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m'\ s\ fd\ sp\ pc'\ rs}$$

(COND)

$$\frac{\begin{array}{c} fd@pc = (cond\text{-}op\ args\ ?\ \texttt{jmp}\ pc_1 : \texttt{jmp}\ pc_2) \\ b = [\![cond\text{-}op]\!]\,(ge, sp, [\![args]\!]\,(rs)) \\ pc' = b\ ?\ pc_1 : pc_2 \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc'\ rs}$$

(CALL1)

$$\frac{\begin{array}{c} fd@pc = (res := f(args)\ \texttt{jmp}\ pc') \\ (l, 0) = [\![f]\!]\,(ge, sp, rs) \\ fds' = \mathrm{findfunc}(ge, l) \\ vs = [\![args]\!]\,(rs) \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{cst}\ m\ ((res, fd, sp, pc', rs){::}s)\ fds'\ vs}$$

(CALL2-INTERNAL)

$$\frac{\begin{array}{c} (m', l) = \mathrm{alloc}(m, \mathrm{stacksize}(fd)) \\ pc = \mathrm{entrynode}(fd) \\ rs = \mathrm{init\text{-}regs}(\mathrm{params}(fd), vs) \end{array}}{\texttt{cst}\ m\ s\ fd\ vs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m'\ s\ fd\ (l, 0)\ pc\ rs}$$

(CALL2-EXTERNAL)

$$\frac{(\sigma, v, m') \in \mathrm{extcall\text{-}sem}(fs, ge, vs, m)}{\texttt{cst}\ m\ s\ fs\ vs \overset{\sigma}{\hookrightarrow}_{ge} \texttt{rst}\ m'\ s\ v}$$

(RETURN1)

$$\frac{\begin{array}{c} fd@pc = (\texttt{return}\ r) \\ v = [\![r]\!]\,(rs) \\ m' = \mathrm{free}(m, sp, \mathrm{stacksize}(fd)) \end{array}}{\texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{rst}\ m'\ s\ v}$$

(RETURN2)

$$\frac{rs' = rs[res \leftarrow v]}{\texttt{rst}\ m\ (res, fd, sp, pc, rs){::}s\ v \overset{\epsilon}{\hookrightarrow}_{ge} \texttt{ist}\ m\ s\ fd\ sp\ pc\ rs'}$$

Figure I.4 Operational semantics of RTL

The corresponding rule calculates the new values of the registers, the memory (for store instructions), and the next program counter. Calls and returns are treated a bit differently: they do not directly transition from an instruction state to the next instruction state—they go through an intermediate call/return state.

In more detail, if the next instruction is a call instruction, rule (CALL1) looks up the function in the global environment, evaluates its arguments, creates a new stack frame corresponding to the current instruction state, and transitions to a call state. From a call state, there are two possible execution steps. If the function to be called is internal, *i.e.*, we have its function definition $fd \in \mathrm{FDef}$, rule (CALL2-INTERNAL) applies. It allocates the necessary stack space for the called function, initializes the parameter registers with the values passed as arguments, sets the program counter to point to the first node of the called function, and moves to the appropriate instruction state of the called function. If the function to be called is external, *i.e.*, we have a function signature $fs \in \mathrm{FSig}$, rule (CALL2-EXTERNAL) goes directly to the return state, and generates an event σ indicating that it called an external function.

Conversely, rule (RETURN1) returns from a function by evaluating the result to be returned, deallocating the stack space used by the function, and transitioning to the return state. Then rule (RETURN2) pops the top-most stack frame and transitions to a normal instruction state thereby restoring the registers, program counter, and stack pointers of the calling function.

2.4 Constant Propagation

Now we explain how CompCert's constant propagation works, and how it verifies the optimization using the simulation technique.

Algorithm Given a program *prg*, constant propagation walks through each function definition fd of the program and transforms it using the function $\mathrm{transfun}(prg, fd)$. This in turn runs a "value analysis" to determine which variables (whether global variables or local registers) hold a known constant value at each program point and then, based on that information, simplifies the program.

The analysis consists of two parts: (a) the global part, which detects which global variables cannot be updated (*i.e.*, those declared with the `const` qualifier), and (b) the local part, which analyzes the code of a function and calculates an abstract value for each register and stack variable. The abstract value of a variable can be either $\bot$ if the variable holds `undef`, or a constant number, or *NS* if the variable contains anything except for a pointer pointing into the current stack frame, or $\top$ if no more precise information is known. These abstract values form a lattice by taking the order $\bot \sqsubseteq num \sqsubseteq NS \sqsubseteq \top$.

The value analysis performs a usual traversal of the code. When calling a function, if it can be determined that no memory address can point to the current stack frame and none of the function's arguments point to the current stack frame (*i.e.*, their abstract value is at most *NS*), then the abstract value of the function's result is also *NS*, and the abstraction of the stack memory is preserved. If, however, a pointer to the current stack frame has escaped, then any information about the stack memory is forgotten.

The transformation part itself is straightforward: at each node n of the function's CFG, if the analysis has determined that a variable has a constant value at node n, then the use of that variable is replaced by the constant it holds, and the instruction is suitably simplified.

For example, the algorithms works for the constant propagation example presented in Section 3 as follows (albeit presented for a different language). First, the analysis result for `x` is the constant number 42 after line 10, and it continues to be 42 after line 20 because the address of `x` is not leaked to the external function. Then based on this analysis result, CompCert replaces `x` with 42 at line 30.

As a more interesting example, Figure I.5 shows the effect of constant propagation applied to a simple program. The program contains one internal function, `f`, which calls an external function, `g`, and three zero-initialized variables: a local variable (a register), `x`, an address-taken variable on the stack, `sp[0]`, and a global variable, `gv[0]`. After the external function call, constant propagation can safely assume that `x` = 0 and thereby simplify the conditional at node 5, but cannot do the same for `sp[0]` at node 4 because its address was passed to the external function and its value might therefore have changed. Further, at node 6, constant propagation notes that the global variable `gv[0]` has been declared with the `const` qualifier, and can therefore assume that `gv[0]` = 0.

Verification The correctness proof of the constant propagation pass in CompCert establishes the existence of a simulation relation, R, that relates the loading of the source and target programs. That is, for every well-formed source RTL program *prg*, it proves there exists a simulation $R(prg)$ such that $(\text{load}(prg), \text{load}(\mathcal{T}_{\text{cp}}(prg))) \in R(prg)$.

The simulation relation, R, used for the constant propagation pass is given in Figure I.6. It is defined in terms of matching relations on states, stack frames, and stacks and function definitions ($\sim_{\text{state}}$, $\sim_{\text{frame}}$, $\sim_{\text{stack}}$, and $\sim_{\text{fdef}}$ respectively). These relations take as a parameter the source program, *prg*, which is used to relate the function definitions of the source and target programs.[5]

[5] The version shown here is a slight simplification of the actual simulation used in the constant propa-

```
extern g(a,b);                          extern g(a,b);
const gv[1] := { 0 };                   const gv[1] := { 0 };

f() {                                   f() {
  x, y;                                   x, y;
  sp[4];                                  sp[4];

1: sp[0] := 0 jmp 2;                    1: sp[0] := 0 jmp 2;
2: x := 0 jmp 3;                        2: x := 0 jmp 3;
3: y := g(sp,x) jmp 4;                  3: y := g(sp,x) jmp 4;
4: sp[0] > 0 ? jmp 5 : jmp 6;           4: sp[0] > 0 ? jmp 5 : jmp 6;
5: x > 0 ? jmp 6 : jmp 7;          ↦    5: jmp 7;            // CHANGED
6: return gv[0];                   ↦    6: return 0;         // CHANGED
7: return y;                            7: return y;
}                                       }
```

Figure I.5 Example of constant propagation

We say that two function definitions are related in the program prg, written $fd \sim_{\text{fdef}} fd'$, if the target function, fd', is the result of applying constant propagation to the source function, fd. Two stack frames are related by $prg \vdash sf \sim_{\text{frame}} sf'$ if the function code in sf' is the transformation of the function code of sf, the stack pointer and program counters agree, and the registers of sf' hold equal or more defined values than those of sf. Two stacks are related, $prg \vdash s \sim_{\text{stack}} s'$, if they have the same length and their stack frames are related element-wise.

Two states are related, $prg \vdash s \sim_{\text{state}} s'$ if (1) they are of the same kind, (2) the memory of s' is an extension of that of s, (3) their stacks are related by $\sim_{\text{stack}}$, (4) the respective function definitions are related by $\sim_{\text{fdef}}$ (when applicable), (5) the stack pointer and program counter agree (when applicable), and (6) the registers/arguments/return value of s is equal or less defined than that of s'.

Finally, the two states are in the simulation relation R if they are related by $\sim_{\text{state}}$ and the source state satisfies the value analysis invariant, sound-state(prg, s). This invariant basically says that the (concrete) value of each variable in the state s is included in the variable's abstract value computed by the analysis at the current program point. The invariant depends on the program for two reasons: (1) so that it can calculate the global environment, $ge = get\text{-}genv(prg)$, and (2) so that it can 'run' the analysis on the program so as to be able to compare its results with the current state.

The basic soundness properties of the value analysis are (1) that the sound-state

gation pass. It abstracts away some tedious details of the actual $\sim_{\text{frame}}$ definition that are orthogonal to our story. This is merely to streamline the presentation.

$$prg \vdash fd \sim_{\text{fdef}} fd' \overset{\text{def}}{=} fd' = \text{transfun}(prg, fd)$$

$$\frac{prg \vdash fd \sim_{\text{fdef}} fd' \qquad rs \leq_{\texttt{def}} rs'}{prg \vdash (r, fd, sp, pc, rs) \sim_{\text{frame}} (r, fd', sp, pc, rs')}$$

$$\frac{}{prg \vdash [\,] \sim_{\text{stack}} [\,]} \qquad \frac{prg \vdash sf \sim_{\text{frame}} sf' \qquad s \sim_{\text{stack}} s'}{prg \vdash sf::s \sim_{\text{stack}} sf::s'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \quad s \sim_{\text{stack}} s' \quad prg \vdash fd \sim_{\text{fdef}} fd' \quad rs \leq_{\texttt{def}} rs'}{prg \vdash \texttt{ist}\ m\ s\ fd\ sp\ pc\ rs \sim_{\text{state}} \texttt{ist}\ m'\ s'\ fd'\ sp\ pc\ rs'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \quad s \sim_{\text{stack}} s' \quad prg \vdash fd \sim_{\text{fdef}} fd' \quad args \leq_{\texttt{def}} args'}{prg \vdash \texttt{cst}\ m\ s\ fd\ args \sim_{\text{state}} \texttt{cst}\ m'\ s'\ fd'\ args'}$$

$$\frac{m \sqsubseteq_{\text{ext}} m' \qquad s \sim_{\text{stack}} s' \qquad v \leq_{\texttt{def}} v'}{prg \vdash \texttt{rst}\ m\ s\ v \sim_{\text{state}} \texttt{rst}\ m'\ s'\ v'}$$

$$(s, s') \in R(prg) \overset{\text{def}}{=} prg \vdash s \sim_{\text{state}} s' \wedge \text{sound-state}(prg, s)$$

Figure I.6 CompCert’s simulation relation for the constant propagation

invariant holds for the initial state of a program, and (2) that it is preserved by execution steps. Formally:

$$\frac{s = \mathit{load}(\mathit{prg})}{\text{sound-state}(\mathit{prg}, s)} \qquad \frac{\begin{array}{c}\text{sound-state}(\mathit{prg}, s) \\ \mathit{ge} = \mathit{get\text{-}genv}(\mathit{prg}) \qquad s \overset{t}{\hookrightarrow}_{\mathit{ge}} s'\end{array}}{\text{sound-state}(\mathit{prg}, s')}$$

The CompCert proof then establishes the following two properties of R, which collectively imply compiler correctness, *i.e.*, the target's behaviors refine the source's:

(1) $(\text{load}(\mathit{prg}), \text{load}(\mathcal{T}_{\text{cp}}(\mathit{prg}))) \in R$.

(2) R is indeed a simulation relation. Specifically, it is a "backward" simulation, meaning that for any related states $(s, t) \in R$, if the target state t takes a step to t' with an event σ, the source s also takes a step to *some* state s' with the same event σ, such that $(s', t') \in R$.

As for (1), the initial states after loading satisfy $\sim_{\text{state}}$ by construction, and the initial source state satisfies sound-state thanks to the soundness of the value analysis above. As for (2), R is indeed a backward simulation for the following reasons. From $(s, t) \in R$, we know that we are executing the instructions at the same pc. Thus by the definition of constant propagation, the target instruction is either identical to the source instruction or obtained by replacing a variable with a constant or converting a conditional jump to an unconditional jump, depending on the value analysis result. Here, thanks to the soundness of the current state w.r.t. the analysis result and the relation between the two states specified by $\sim_{\text{state}}$, we can easily deduce that executing the source and target instructions results in related states. Also, the soundness of the new source state s' follows from the soundness preservation property of the value analysis stated above.

Note: The verification approach described above, relying on backward simulations, is something of an oversimplification of what CompCert actually does. In fact, to make the proofs more convenient, CompCert uses "forward" simulations for the backend passes. We briefly discuss why CompCert uses forward simulation, even though it implies that the source's behaviors refine the target's, which seems the wrong way around. First, a forward simulation is easier to establish than a backward one because a single instruction in the source may be compiled down to several instructions in the target. Second, CompCert composes forward simulations of backend passes using the transitivity of forward simulations, which is not hard to show. Then it converts the composed forward simulation between an IR and assembly to a backward simulation between them using some technical properties of the IR and assembly (namely, that the IR is "receptive" and assembly is "determinate"). Finally, from this backward simulation, one can establish that the target's behaviors refine the source's.

So far we discussed the technical background that is needed to understand the main contributions of this dissertation, which we will present from now on. We will start with a formal semantics of casts between integers and pointers.

Chapter II

Relaxed-Memory Concurrency

3 Introduction

What is the right semantics for concurrent shared-memory programs written in higher-level languages? For programmers, the simplest answer would be a *sequentially consistent (SC)* semantics, in which all threads in a program share a single view of memory and writes to memory take immediate global effect.

However, a naive SC semantics is costly to implement. First of all, commodity architectures (such as x86, Power, and ARM) are not SC: they execute memory operations speculatively or out of order, and they employ hierarchies of buffers to reduce memory latency, with the effect that there is no globally consistent view of memory shared by all threads. To simulate SC semantics on these architectures, one must therefore insert expensive fence instructions to subvert the efforts of the hardware. Secondly, a number of common compiler optimizations—such as constant propagation—are rendered unsound by a naive SC semantics because they effectively reorder memory operations. Moreover, SC semantics is stronger (*i.e.*, more restrictive) than necessary for many concurrent algorithms.

Hence, languages like C/C++ and Java have opted instead to provide *relaxed* (aka *weak*) memory models [59, 36], which enable programmers to demand SC semantics when they need it, but which also support a range of cheaper memory operations that trade off strongly consistent and/or well-defined behavior for efficiency.

3.1 Criteria for a Programming Language Memory Model

Unfortunately, despite many years of research, it has proven very difficult to develop a memory model for concurrent programming languages that adequately balances the conflicting desiderata of programmers, compilers, and hardware. In particular, we would like to find a memory model that satisfies the following properties:

- The model should support *high-level reasoning* principles that programmers and compiler analyses depend on. At a bare minimum, it should validate simple invariant-based verification, and should provide some "DRF" guarantees [10], ensuring that programmers who employ sufficient synchronization need not understand the full complexities of relaxed-memory semantics.

- The model should be *implementable*, *i.e.*, it should validate common compiler optimizations, as well as standard compilation schemes to the major modern architectures. To be implementable, it must justify many kinds of instruction reordering and merging.

- The model should ideally *avoid relying on undefined behavior* to define the semantics of racy programs. This is a prerequisite for applicability to type-safe languages like Java, in which well-typed programs may contain data races but are nevertheless expected to have safe, well-defined semantics.

Both C/C++ and Java fail to achieve some of these criteria.

In the case of Java, the memory model fails to validate a number of common program transformations performed by real Java compilers, such as redundant read-after-read elimination and "roach motel" reordering [72]. Although this problem has been known for some time, a satisfactory solution has yet to be developed.

In the case of C/C++, the memory model relies crucially on undefined behaviors to give semantics to racy programs. Moreover, it permits certain "out-of-thin-air" executions which violate basic invariant-based reasoning (and DRF guarantees) [20].

3.2 The "Out of Thin Air" Problem

To illustrate the problem with C/C++, consider these two variants of the classic "load buffering" litmus test (with two threads in parallel):

$$\left.\begin{array}{l} a := x; \\ y := 1; \end{array}\right\| \; x := y; \qquad \text{(LB)} \qquad\qquad \left.\begin{array}{l} a := x; \\ y := a; \end{array}\right\| \; x := y; \qquad \text{(LBd)}$$

Here, we assume that all variables are initially 0, and that all memory accesses are of the weakest consistency level, *i.e.*, they are compiled down to plain loads and stores at the hardware level with no additional synchronization (in C/C++ this is called "relaxed"). The question is: should it be possible for these programs to assign 1 to a? In the case of LB, the answer is yes: architectures like Power and ARM may reorder the write of y before the read of x in the first thread (since these are accesses to distinct variables), after which a can be assigned 1 by a standard interleaving execution. In the case of LBd, however, the answer *ought* to be no: all the operations simply copy one variable to another and all are initially 0, so if a could receive 1, it would come "out of thin air". No hardware reorderings or reasonable compiler optimizations will produce this behavior. If they did, it would cause major problems: one would not be able to establish even basic invariants (such as $x = y = 0$), and basic sanity results like the aforementioned DRF theorems would cease to hold. It is therefore a serious problem that the formal memory model of C/C++ allows such out-of-thin-air (OOTA) behavior.

Intuitively, the reason C/C++ allows OOTA behaviors is that it is not clear how to distinguish them from acceptable behaviors. The C/C++ model formalizes valid executions as graphs of memory access events (think: partially-ordered traces) subject to a set of coherence axioms, and the same coherent event graph that describes a valid execution of LB in which a receives 1 also describes a valid execution of LBd in which a receives 1.

Hardware memory models (*e.g.*, Power and ARM) handle this problem by taking syntactic dependencies between instructions into account in determining program semantics. Under such models, the out-of-order execution in LB is valid because the write to y is independent of the read from x, whereas in LBd such out-of-order execution is prevented by the syntactic dependency between the two instructions. Although this approach is suitable for modeling hardware, it is too brittle for a language-level semantics because it fails to validate standard compiler optimizations that remove syntactic dependencies (see also [20]). As a very simple example, consider the following variant of LB and LBd:

$$\begin{array}{l} a := x; \\ y := a + 1 - a; \end{array} \;\left\|\; x := y; \right. \tag{LBfd}$$

Under the hardware models, this LBfd program would be treated similarly to LBd due to the syntactic data dependency, so a could not receive 1. But even a basic optimizing compiler could trivially transform LBfd to LB, in which case a could receive 1.

As a result, we still to this day lack a semantics for relaxed-memory concurrency in C/C++ and Java that corresponds to how these languages are implemented and that provides sufficient reasoning guarantees to programmers and compiler-writers. Several proposals have recently been made for how to fix the C/C++ and Java memory models (some of which are discussed in Section 9), but none have been proven to validate the full range of standard optimizations/reorderings performed by C/C++ and Java compilers and by commodity hardware like Power and ARM. Furthermore, for most of the existing proposals, it is known that indeed they do *not* validate some important reorderings.

3.3 A "Promising" Semantics for Relaxed Memory

In this paper, we present what we believe is a very promising way forward: the first relaxed memory model to support a broad spectrum of features from the C/C++ concurrency model while also satisfying all three criteria listed in Section 3.1.

We achieve these ends through a combination of mechanisms (some standard, some not), but the most important and novel idea for the reader to take away from this paper is the notion of *promises*.

Under our model, which is defined by an operational semantics, a thread T may nondeterministically "promise" to write a value v to a memory location x at some point in the future. From the point of view of other threads, a promise is no different from an ordinary write: once T has promised to write v to x, other threads can read from that write. (In contrast, T cannot read from its own promised write until T has fulfilled the promise: this is crucial to preserve basic sanity of the semantics.) Intuitively, promises simulate the effect of read-write reorderings by allowing write events to be visible to other threads before the point at which they occur in the program order.

We must, however, ensure that promises do not introduce bad OOTA behaviors. Toward this end, we only allow T to promise to write v to x if it is possible to *thread-locally certify* that the promise can be fulfilled in a finite number of steps. That is, we must show that T will be able to write v to x after some finite sequence of steps of T's execution (*i.e.*, with no help from other threads). The certification requirement guarantees absence of bad OOTA executions by ensuring that T can only promise to write a value v to x if T could have written v to x anyway.

Returning to the examples from Section 3.2, it is easy to see how promises give us the desired semantics:

- In LB, the first thread can promise to write 1 to y (since it will indeed write 1 to y no matter what value is assigned to a), after which the second thread can read from that promise and write 1 to x. Subsequently, the first thread can execute normally, reading 1 from x and assigning it to a.

- The execution of LBfd may proceed in exactly the same way. The fact that the write of y depends syntactically on a is irrelevant, because during certification of the promised write of 1 to y, the expression $a + 1 - a$ will always evaluate to 1.

- By contrast, the OOTA behavior will not be allowed for LBd. In order for the first thread to promise to write 1 to y, it would need to certify that it can write 1 to y without promises. But since all variables are initially 0, this is not possible.

Our model supports all features of C/C++ concurrency except consume reads and SC accesses. Consume reads are widely considered a premature aspect of the C18/C++17 standard and are currently implemented the same as acquire reads in mainstream compilers. In contrast, SC accesses are a major feature of C/C++, and originally our model included an account of SC accesses as well. However, in the course of trying to prove the correctness of compilation to Power, we discovered that our semantics of SC accesses was flawed, and this led us to discover a flaw in the C/C++11 standard as well! (See [53] for further details.) Thus, a proper handling of SC accesses remains an open and important problem for future work.

In the rest of the paper, we will flesh out the idea of promises—as well as the other elements of our model—in layers. We begin in Section 4 by presenting the details of our model restricted to relaxed reads and writes. In Section 5, we extend this base model further to support atomic updates (*i.e.*, read-modify-write operations, like CAS). Then, in Section 6, we scale the model up to handle most features of the C/C++ memory model. In Section 7, we present our formal results—validating many program transformations, compilation to x86-TSO, DRF theorems, and an invariant-based logic—most of which are fully mechanized in the Coq proof assistant (totalling about 37K lines of Coq). In Section 9, we compare with related work, and in Section 10, we conclude with discussion of future work.

4 Basic Model for Handling Relaxed Accesses

In this section, we introduce the key ideas of our memory model, first by example and then more formally. At first we will only consider a semantics for fully "relaxed" atomic read and write accesses (in the sense of C/C++). This is a natural starting point, since the OOTA problem is fundamentally about how to give a reasonable semantics for these relaxed accesses, and the key elements of our solution are easiest to understand in this simpler setting. We will see in subsequent sections how to extend and generalize this base model to account for a much richer variety of memory operations.

To illustrate our semantics, we will write small programs such as the following:

$$\begin{array}{l||l} x := 1; & y := 1; \\ a := y; \ /\!/ \ 0 & b := x; \ /\!/ \ 0 \end{array} \tag{SB}$$

As a convention, we write a, b, c for local variables (registers) and x, y, z for (distinct) shared memory locations, and assume that all variables are initialized to 0. We refer to thread i as T_i. Moreover, in order to refer to a specific observation of the program, we annotate the corresponding reads with the values expected to be read (*e.g.*, in the above program, the comment notation indicates the observed result that $a = b = 0$).

4.1 Main Ideas

High-Level Requirements: Reorderings and Coherence Relaxed read and write operations are intended to be compiled down directly to plain loads and stores at the machine level, so one of the main requirements of our semantics is that it be at least as permissive as commodity hardware. Toward this end, our semantics must justify reordering of independent memory operations (*i.e.*, operations that access distinct locations), since the more weakly consistent architectures (like ARM) may potentially perform such reorderings. There are four such classes of reorderings—write-read, write-write, read-read, and read-write—and in Section 7 we will prove formally that our semantics justifies all of them.

On the other hand, it is also important that our semantics not be unnecessarily weak. In particular, all the existing implementations of C/C++, even for weaker architectures like Power and ARM, guarantee at a bare minimum a property we call *per-location coherence* (aka *SC-per-location*). Per-location coherence says that, even though threads may observe writes to different locations in different orders, they must observe writes to the *same* location in a single total order (called the "modification order" in C/C++ lingo). In addition to being supported by hardware, per-location coherence is preserved by common compiler optimizations as well. Hence, we want our semantics of relaxed accesses to guarantee it. (In Section 6.3 we will present an even weaker mode of accesses that does not provide full per-location coherence.)

Operational Semantics with Timestamps In contrast to the C/C++ memory model, which relies on declarative semantics over event graphs, ours employs a more standard SC-style operational semantics for concurrency, in which the executions of different threads are nondeterministically interleaved. However, in order to account for weak memory behaviors, we use a more elaborate memory representation than the standard

SC semantics does. Instead of being a flat map from addresses to values, our memory records the set of all writes ever performed. It may help to think of writes as messages, and memory as a message pool which grows monotonically. When a thread T reads from a location x, it need not read "the latest" write to x, since there is no shared understanding among threads of what the latest write is. The thread T thus retains flexibility in terms of which message it reads, but we must place some restrictions on this flexibility in order to guarantee per-location coherence.

Specifically, we totally order the writes to each location by attaching a (unique) *timestamp* to each write message. Thus, messages are triples of the form $\langle x : v@t \rangle$ (where x is a location, v a value, and t a timestamp). (The *modification order* for a location x is thus implicitly derivable from the order of timestamps on x's messages.) In addition, for each thread T, we keep track of a map from locations x to the largest timestamp of a write to x that T has observed or executed. We refer to this map as T's *view* of memory, and one can think of it as recording the set of most recent write messages that T has observed. Hence, when T reads from a location x, it must read from a message with a timestamp *at least as large* as the one recorded for x in T's view. And when T writes to x, it must pick a timestamp *strictly larger* than the one recorded for x in its view.

Let us see now how our semantics, as explained thus far, already suffices to justify desirable reorderings while ruling out violations of coherence. First, recall the write-read reordering exhibited by the "store buffering" SB example above, and let us see how the behavior can be justified. Initially, assume the memory contains the initialization messages $\langle x : 0@0 \rangle$ and $\langle y : 0@0 \rangle$, and both threads maintain a view of x and y that maps them to 0. When T_1 performs the assignment $x := 1$, it will choose some timestamp $t > 0$, add the message $\langle x : 1@t \rangle$ to the memory, and update its view of x to t. But this will have no effect on its view of y or T_2's view of x, which remain at 0. Thus, when T_1 subsequently reads y, it is free to read 0. (And analogously for T_2.)

On the flip side, per-location timestamps also explain why the following coherence violation is forbidden.

$$\begin{array}{l||l} x := 1; & x := 2; \\ a := x; \ /\!/\ 2 & b := x; \ /\!/\ 1 \end{array} \tag{COH}$$

Here, the two writes to x must be totally ordered. Suppose, without loss of generality, that the $x := 1$ was written at timestamp 1 and $x := 2$ at timestamp 2. Then, although T_1 may read value 2, T_2 cannot read 1, because 1 was written at a smaller timestamp than the one that T_2 already recorded in its view when it wrote $x := 2$.

One subtle point is that, when writing to a location x, a thread T may select any unused timestamp t larger than the one recorded in its view of x, but t need not be globally maximal. That is, t may be smaller than the timestamp that another thread has already used for a write to x. This freedom is in fact crucial in order to permit write-write reorderings, as exemplified by the following test case:

$$\begin{array}{l||l} x := 2; & y := 2; \\ y := 1; & x := 1; \\ a := y;\ \textit{// 2} & b := x;\ \textit{// 2} \end{array} \tag{2+2W}$$

To get the desired weak outcome, the writes of $x := 1$ and $y := 1$ must pick smaller timestamps than the $x := 2$ and $y := 2$ writes, respectively, but at least one of the 1-writes must be executed *after* the 2-write to the same location. Thus, it is essential to be able to write using a timestamp that is not globally maximal.

Promises Unfortunately, our timestamp semantics alone does not suffice to explain *read-write* reorderings, as exemplified by the (LB) and (LBfd) programs from Section 3.2. It is precisely these reorderings that motivate our introduction of *promises*.

As explained in Section 3.3, a thread T may at any point *promise* to write $x := v$ at some timestamp t (provided that t is greater than T's current view of x). This promise is treated to a large extent like an actual write operation. In particular, it adds a new message $\langle x : v@t \rangle$ to memory, which may then be read by other threads. However, in order to make such a promise, T must *thread-locally certify* it—that is, T must demonstrate that it will be able to fulfill this promise (writing $x := v$ at timestamp t) in a finite number of thread-local steps. Certification is needed to guarantee plausibility of the promise, but crucially, there is no requirement that the specific steps of execution taken during certification must match the subsequent steps of actual execution. Indeed, we already witnessed this with the (LB) and (LBfd) executions, where T_1 read $x = 0$ during the initial certification of its promised write to y, but read $x = 1$ during the actual execution.

Let us now briefly touch on a few technical points concerning the interaction of promises and timestamps.

First of all, it is important that T cannot directly read its own promises, because this would violate per-location coherence: for example, the single-threaded program $a := x;\ x := 1$ would be able to return $a = 1$! Note that we do not need to explicitly enforce this restriction—it just falls out from our rules concerning timestamps. In particular, if T were to promise $\langle x : v@t \rangle$, and then were to read from its own promise, then T's view of x would be updated to t, and there would be no way for T to subsequently fulfill the promise because it would have to pick a timestamp strictly greater than t when performing the assignment $x := v$.

(THREAD: SILENT)

$$\dfrac{\sigma \xrightarrow{\text{Silent}} \sigma'}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V, P\rangle, M\rangle}$$

(THREAD: READ)

$$\dfrac{\sigma \xrightarrow{\mathtt{R}(x,v)} \sigma' \quad \langle x : v@t\rangle \in M \\ V(x) \le t \quad V' = V[x \mapsto t]}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P\rangle, M\rangle}$$

(THREAD: WRITE)

$$\dfrac{\sigma \xrightarrow{\mathtt{W}(x,v)} \sigma' \quad M' = M \overset{\scriptscriptstyle A}{\hookleftarrow} \langle x : v@t\rangle \\ V(x) < t \quad V' = V[x \mapsto t]}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P\rangle, M'\rangle}$$

(THREAD: PROMISE)

$$\dfrac{M' = M \overset{\scriptscriptstyle A}{\hookleftarrow} m \qquad P' = P \overset{\scriptscriptstyle A}{\hookleftarrow} m}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma, V, P'\rangle, M'\rangle}$$

(THREAD: FULFILL)

$$\dfrac{\sigma \xrightarrow{\mathtt{W}(x,v)} \sigma' \quad \langle x : v@t\rangle \in P \quad P' = P \setminus \{\langle x : v@t\rangle\} \\ V(x) < t \quad V' = V[x \mapsto t]}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P'\rangle, M\rangle}$$

(MACHINE STEP)

$$\dfrac{\langle \mathcal{TS}(i), M\rangle \to^{+} \langle TS', M'\rangle \\ \langle TS', M'\rangle \text{ is consistent}}{\langle \mathcal{TS}, M\rangle \to \langle \mathcal{TS}[i \mapsto TS'], M'\rangle}$$

Figure II.1 Operational semantics for only relaxed read and write

That said, it is possible for T to read its promised value indirectly via another thread, as in the LB and LBfd programs. It may even read the promised value from the same location where it promised to write it, as in the following example [52].

$$\begin{array}{l||l||l} a := x; \ /\!/\ 1 & & \\ & y := x; & x := y; \\ x := 1; & & \end{array} \qquad \text{(ARM-weak)}$$

This outcome can be explained by T_1 promising $\langle x : 1@2\rangle$, then T_2 reading $x = 1$ and storing it to y, and T_3 reading $y = 1$ and writing $x := 1$ at timestamp 1, which T_1 can read before fulfilling its promise. Such behavior, strange though it may seem, is actually allowed (though not yet observed) by the ARM memory model [27].

Last but not least, we wish to ensure that promises do not lead to impossible situations later down the road, *i.e.*, that making a promise cannot cause the execution of a program to get stuck. The thread-local certification that accompanies a promise step goes some way toward ensuring this progress condition, but it is not enough. We also amend the semantics in the following two ways:

1. Every step a thread takes, it must *re-certify* all its outstanding promises to make sure they can *still* be fulfilled. To see why, consider a possible execution of the following program:

$$\begin{array}{l||l} a := x; & \\ & x := 2; \\ x := 1; & \end{array}$$

Suppose that T_1 (for no particularly good reason) promises $\langle x : 1@1 \rangle$. At first, this is easy to certify: T_1 can read the initial value of x (the message $\langle x : 0@0 \rangle$), and then perform the assignment $x := 1$ picking timestamp 1. Suppose then that T_2 picks the timestamp 2 when performing $x := 2$. If at this point in the execution T_1 were permitted to read the message $\langle x : 2@2 \rangle$, it would have the effect of bumping up T_1's view of x to timestamp 2, which would prevent it from subsequently fulfilling its promise. It is thus crucial that T_1 *not* be allowed to read $x = 2$ (in this particular execution), and indeed our semantics will not allow it to do so because the re-certification check would fail. As the example illustrates, promises can restrict a thread's future nondeterministic choices concerning the messages it reads.

2. We require the total order on timestamps to be *dense* (*e.g.*, choosing timestamps to be rational numbers), so that there is always a place to put intermediate writes before a promise. Consider, for example, the following program:

$$\begin{array}{r||l} x := 1; & \\ & x := 3; \\ x := 2; & \end{array}$$

Here, T_1 may promise $\langle x : 2@2 \rangle$—when validating this promise, T_1 might write $\langle x : 1@1 \rangle$ before writing $\langle x : 2@2 \rangle$. If, however, T_2 subsequently writes $\langle x : 3@1 \rangle$ before T_1 has actually written $x := 1$, then T_1 can no longer pick 1 as a timestamp for $x := 1$. To make progress here, T_1 needs a timestamp for $x := 1$ strictly between 0 and 2, and 1 is already taken. By requiring the timestamp order to be dense, we ensure that there is always some free timestamp (*e.g.*, 1.5) that T_1 can use.

4.2 Formal Definition

We now define our model for relaxed accesses formally. Let Loc be the set of memory locations, Val be the set of values, and Time be an infinite set of *timestamps*, densely totally ordered by $\leq$, with 0 being the minimum element. A *timemap* is a function $T : \mathsf{Loc} \to \mathsf{Time}$. The order $\leq$ is extended pointwise to timemaps.

Programming Language To keep the presentation abstract, we do not fix a particular programming language; we simply consider each thread i as a transition system with a set of states State_i, initial state $\sigma_i^0 \in \mathsf{State}_i$ and final state $\sigma_i^{\text{final}} \in \mathsf{State}_i$. Intuitively, these states store the values of the local registers and the program counter. Transitions are labeled: the label $\mathsf{R}(x, v)$ corresponds to a transition that reads the value v from location

x, and $\mathtt{W}(x, v)$ denotes a write of the value v to x, while local transitions that do not access the memory are labeled with "Silent". We assume *receptiveness* of the transition systems—whenever an $\mathtt{R}(x, v)$-transition is possible from a state σ_i, so is an $\mathtt{R}(x, v')$-transition for every value v'—and that they only get stuck in the σ_i^{final} states.

Messages A *message* m is a tuple $\langle x : v@t \rangle$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$ and $t \in \mathsf{Time}$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, and $m.\mathtt{t}$ the components of a message m. Two messages m and m' are called *disjoint*, denoted $m \# m'$, if $m.\mathtt{loc} \neq m'.\mathtt{loc}$ or $m.\mathtt{t} \neq m'.\mathtt{t}$. Two sets M and M' of messages are called *disjoint*, denoted $M \# M'$, if $m \# m'$ for every $m \in M$ and $m' \in M'$.

Memory A *memory* is a pairwise disjoint finite set of messages. A message m may be *(additively) inserted* into memory M if m is disjoint from every message in M. Formally, the additive insertion $M \overset{\mathrm{A}}{\hookleftarrow} m$ is given by $M \cup \{m\}$ and is only defined if $M \# \{m\}$.

Thread States and Configurations A *thread state* is a triple $TS = \langle \sigma, V, P \rangle$, where σ is the thread's local state, V is a timemap representing the thread's *view* of memory, and P is a memory that keeps track of the thread's outstanding promises. We denote by $TS.\mathtt{st}$, $TS.\mathtt{view}$, and $TS.\mathtt{prm}$ the components of a thread state TS. In turn, a *thread configuration* is a pair $\mathbf{TC} = \langle TS, M \rangle$, where TS is a thread state and M is a memory, called the *global memory*. Note that we will always have $TS.\mathtt{prm} \subseteq M$.

Figure II.1 shows the five reduction rules for thread configurations. The SILENT rule handles the case when the program performs some local computation that does not affect memory. The READ rule handles the case when the program reads from a location x. The rule nondeterministically selects some message m in the memory, whose timestamp is greater than or equal to the timestamp recorded for x in the thread's view, and returns its value; it also updates the thread's view of x to the timestamp of m. The WRITE rule handles the case when the program writes to location x. It extends the memory with a new message for x, whose timestamp t is greater than the one recorded for x in the thread's view, and it updates the thread's view of x to match t. The PROMISE rule extends the memory and the thread's promise set with an arbitrary new message m, whose timestamp is not already present in the memory. (The promise certification is handled separately, as described below.) Finally, the FULFILL rule is similar to the WRITE rule, except that instead of adding a message to the memory, it removes an appropriate message from the thread's promise set P.

We note that the WRITE rule is redundant; we merely included it to improve readability. Any application of WRITE can be simulated by first promising the appropriate message with the PROMISE rule and then immediately fulfilling the promise with the FULFILL rule.

As we have already mentioned, we have to restrict thread executions so that all promises a thread makes are fulfillable. Thread configurations satisfying this property are called *consistent*. Formally, a thread configuration $\langle TS, M\rangle$ is *consistent* if $\langle TS, M\rangle \rightarrow^* \langle TS', M'\rangle$ for some TS' and M' such that $TS'.\mathtt{prm} = \varnothing$. Notice that in the certification of a promise, it is formally possible to make further promises. Since, however, in the end all such promises must be fulfilled, it is useless to make such promises. (A proof of this property is included in our formal development.)

Machine States A *machine state* $\mathbf{MS} = \langle \mathcal{TS}, M\rangle$ consists of a function $\mathcal{TS}$ assigning a thread state to every thread, and a (global) memory M. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state σ_i^0, a current timestamp of 0 for every location, and an empty set of promises; and the initial memory M^0 that has one initial message $\langle x : 0@0\rangle$ for each location x. A machine takes a step (see the last rule in Figure II.1) whenever a thread can take several steps to some consistent configuration. Note that we allow multiple thread steps in one machine step. This is convenient in our proofs, and can reduce the number of certifications during an execution of a program.

Finally, we can easily show that a machine never gets stuck unless each thread i has reached $\langle \sigma_i^{\mathrm{final}}, V, \varnothing\rangle$ for some view V. For non-final states, progress follows from the receptiveness and progress assumptions about the programming language, together with the invariant that no thread has a higher view of any x than the highest timestamp for x in memory. Another crucial invariant is consistency: the MACHINE STEP rule demands that each machine step taken by a thread must preserve consistency of the thread's own configuration, and it implicitly preserves the consistency of other threads' configurations as well, since they are free to ignore any new messages the thread has added. When all threads reach their final states, consistency implies they must have no promises left to fulfill.

5 Supporting Atomic Updates

In this section, we extend our basic model for relaxed accesses to also handle (relaxed) *atomic update*—aka *read-modify-write* (RMW)—instructions, such as fetch-and-add and compare-and-swap. Updates are essential as a means to implement synchronization (*e.g.*, mutual exclusion) between threads, but this also makes them tricky to model semantically. In particular, a successful update operation performed by one thread will often have the effect of "winning a race" and hence blocking (previously possible) update operations performed by other "losing" threads. This stands in contrast to the updates-free fragment in Section 4, in which threads are free to ignore the messages of other threads. Thus, to extend our model to support updates, we must ensure that threads performing updates cannot invalidate the already-certified promises of other threads.

An update is an atomic composition of a read and a write to the same location x. However, unlike under SC, atomicity requires more than just avoiding interference of other threads between the two operations. Consider the following example (taking $\mathrm{FAA}(x, 1)$ to be an atomic *fetch-and-add* of 1 to x, which returns the value of x before the increment):

$$a := \mathrm{FAA}(x, 1); \;\Big\|\; b := \mathrm{FAA}(x, 1); \qquad \text{(Par-Inc)}$$

Atomicity ensures that it is not possible for both threads to increment x from 0 to 1 (we must either get $a = 1$ or $b = 1$). To obtain this, we require that the *read timestamp* of the update (*i.e.*, the timestamp of the write message that the update reads from) immediately precede its *write timestamp* (*i.e.*, the timestamp of the write message that the update generates) in x's modification order, and that future writes to x may not be assigned timestamps in between them. In the example above, if both of the updates were to increment x from 0 to 1, the write timestamp for one of the updates would have to come between the read and write timestamps for the other update.

To enforce this restriction, we extend messages to store a continuous range of timestamps rather than a single timestamp. Thus, messages are now tuples of the form $\langle x : v@(f, t]\rangle$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, and $f, t \in \mathsf{Time}$ satisfying $f < t$ or $f = t = 0$. We write $m.\mathtt{from}$ and $m.\mathtt{to}$ to denote the f and t components of a message m. Intuitively, m can be thought of as *reserving* the timestamps in the range $(m.\mathtt{from}, m.\mathtt{to}]$; among these, $m.\mathtt{to}$ is the "real" timestamp of m, but the remaining timestamps in the range are reserved so that other messages cannot use them. Timestamp reservation is reflected in the following revised definition of message disjointness, which enforces that disjoint messages for the same location must have disjoint ranges:

$$\langle x : v@(f, t]\rangle \# \langle x' : v'@(f', t']\rangle \;\triangleq\; x \neq x' \;\lor\; t \leq f' < t' \;\lor\; t' \leq f < t$$

With timestamp reservation, we can easily ensure that the write timestamp of an update is adjacent to its read timestamp in the modification order. Formally, we will say two messages m and m' are *adjacent*, denoted $\mathsf{Adj}(m, m')$, if $m.\mathtt{loc} = m'.\mathtt{loc}$ and $m.\mathtt{to} =$

m'`.from`. In defining the semantics of updates, we will then insist that the message that the update inserts into memory must appear adjacently after the message that it reads from. This suffices to guarantee the correct outcome in the Par-Inc program above.

Although the introduction of timestamp reservation enables us to easily model updates, it creates a complication for promises, namely that timestamp reservations may invalidate the promise certifications already performed by other threads. Consider, for example, the following program:

$$\begin{array}{l||l||l} a := x;\ /\!/\ 1 & & \\ b := \mathrm{FAA}(z,1);\ /\!/\ 0 & x := y; & \mathrm{FAA}(z,1); \\ y := b + 1; & & \end{array} \tag{Upd-Stuck}$$

This behavior ought to be allowed, since hardware could reorder the read of x after the independent accesses to z and y. To produce this behavior, following our semantics from the previous section, T_1 could promise to write $y := 1$ because it can thread-locally certify that the promise can be fulfilled (the certification will involve updating z from 0 to 1). If, however, T_3 then updates z from 0 to 1, that will mean that T_1 can no longer perform the update it needs to fulfill its promise, and its execution will eventually get stuck.

To avoid such stuck executions, we strengthen the check performed by promise certification, *i.e.*, the consistency requirement on thread configurations. We require that each thread's promises are locally fulfillable not only in the current memory, but also *in any future memory*, *i.e.*, any extension of the memory with additional messages. This quantification over future memories ensures that thread configurations remain consistent whenever another thread performs an execution step, and thus the machine cannot get stuck.

Returning to the above example, T_1 will not be permitted to promise to write $y := 1$ in the initial state, precisely because that promise could not be fulfilled under an arbitrary future memory (*e.g.*, one containing the update of T_3, as we showed). T_1 may, however, first promise $\langle z : 1@(0, 1]\rangle$, reserving the time range from the initialization of z up to its increment. T_1 can fulfill that promise, because no future extension of the memory will be able to add any messages in between. After making that promise, T_1 may then promise, *e.g.*, $\langle y : 1@(3, 4]\rangle$, which it can now fulfill under any extension of the memory. With these promises in place, T_3 will be prevented from updating z from 0 to 1; it will be forced to update z from 1 to 2, which will not block the future execution of T_1.

Our quantification here over *all* future memories may seem rather restrictive in that it completely ignores what can or cannot happen in a particular program. That said, we find it a simple and natural way of ensuring "thread locality". The latter is a guiding

(THREAD: FULFILL UPDATE)

$$\frac{\begin{array}{c}\sigma \xrightarrow{\mathsf{U}(x,v_{\mathrm{r}},v_{\mathrm{w}})} \sigma' \quad \langle x : v_{\mathrm{r}}@(f_{\mathrm{r}}, t_{\mathrm{r}}]\rangle \in M \\ m_{\mathrm{w}} = \langle x : v_{\mathrm{w}}@(t_{\mathrm{r}}, t_{\mathrm{w}}]\rangle \quad m_{\mathrm{w}} \in P \quad P' = P \setminus \{m_{\mathrm{w}}\} \\ V(x) \leq t_{\mathrm{r}} \quad V' = V[x \mapsto t_{\mathrm{w}}]\end{array}}{\langle\langle \sigma, V, P\rangle, M\rangle \to \langle\langle \sigma', V', P'\rangle, M\rangle}$$

Figure II.2 Additional rule for updates

principle in our semantics, according to which the set of actions a thread can take is determined only by the current memory and its own state.

Formally, we say that M_{future} is a *future memory* of M if $M_{\text{future}} = M \hookleftarrow m_1 \hookleftarrow \ldots \hookleftarrow m_n$ for some $n \geq 0$ and messages $m_1, \ldots, m_n$. And we now say a thread configuration $\langle TS, M\rangle$ is *consistent* if, for every future memory M_{future} of M, there exist TS' and M' such that $\langle TS, M_{\text{future}}\rangle \to^* \langle TS', M'\rangle$ and $TS'.\mathtt{prm} = \varnothing$.

Finally, we extend the operational semantics for thread configurations with one additional rule for update fulfillment shown in Figure II.2. (All other rules are as before except all messages $\langle x : v@t\rangle$ are replaced by $\langle x : v@(f, t]\rangle$.) This rule forces its write to be adjacent in modification order to its read. As with ordinary writes, a normal (non-promised) update step can be simulated by a promise step immediately followed by fulfillment. Note that the other rules remain exactly the same; they simply ignore the $m.\mathtt{from}$ component of messages m.

6 Full Model

In this section, we extend the basic model of Section 4-5 to handle all the features of the C/C++ concurrency model except SC accesses and consume reads.

6.1 Release/Acquire Synchronization

Release/Acquire Fences A crucial feature of the C/C++ model is the ability for threads to synchronize using memory fences or stronger kinds of atomic accesses. Consider the message-passing test case:

$$\begin{array}{l||l} x := 1; & a := y; \;\; /\!/\, 1 \\ \textbf{fence-rel}; & \textbf{fence-acq}; \\ y := 1; & b := x; \;\; /\!/ \neq 0 \end{array} \qquad \text{(MP+fences)}$$

The release fence between the writes, together with the acquire fence between the reads, prevents the weak behavior of the example (*i.e.*, that of returning $a = 1$ and $b = 0$). Roughly speaking, the C/C++ model forbids this behavior by requiring that whenever a read before an acquire fence reads from a write after a release fence, the two fences synchronize, which in turn means that any write that happens-before the release fence must be visible to any read that happens-after the acquire fence. So, if T_2 reads $y = 1$, then after the acquire fence it *must* read $x = 1$.

To implement this semantics, we extend our model in two ways.

First, we refine each thread's view. Rather than having a single view of which messages it has observed, a thread now has three views: $\mathcal{V} = \langle cur, acq, rel \rangle$. We denote by $\mathcal{V}.\mathtt{cur}$, $\mathcal{V}.\mathtt{acq}$ and $\mathcal{V}.\mathtt{rel}$ the components of a thread view $\mathcal{V}$. A thread's *current view*, `cur`, is as before: it records which messages the thread has observed and restricts which messages a read may return and a write may create. Its *release view*, `rel`, records what the thread's `cur` view *was* at the point of its last release fence. Dually, its *acquire view*, `acq`, records what the thread's `cur` view *will become* if it performs an acquire fence. Consequently, the views are related as follows: $\mathtt{rel} \leq \mathtt{cur} \leq \mathtt{acq}$.

Second, we extend write messages to record a *message view* R, which records the release view of the writing thread at the time the write occurred (updated to include the write itself). Thus, a message now takes the form $m = \langle x : v@(f, t], R \rangle$, where $R(x) = t$. We write $m.\mathtt{view}$ for the message view of m.

During execution of relaxed accesses, a thread's views drift apart. When a thread reads a message, it incorporates the message's view into the thread's `acq` view, but not into its `cur` or `rel` views. When a thread writes a message, it uses the thread's `rel` view as the basis for the message's view, but only incorporates the message itself into the thread's `cur` and `acq` views, not its `rel` view.

Fence commands bring these diverging views closer to one another. Specifically, an acquire fence increases the thread's `cur` view to match its `acq` view, thereby ensuring that the thread is up to date with respect to views of all the messages read before the fence. Symmetrically, a release fence increases the thread's `rel` view to match its `cur` view, thereby ensuring that the views of all messages the thread writes after the release fence will contain the messages observed before the fence.

Returning to the MP+fences program, suppose that T_1 emitted messages $\langle x : 1@(\_, t_x], \_ \rangle$ and $\langle y : 1@(\_, t_y], R_y \rangle$. Then, T_1's `cur` view before the release fence maps x to t_x. The fence then updates T_1's `rel` view to match its `cur` view, so that the message view accompanying the subsequent write to y will map x to t_x as well. (Without the release fence, this message view would map x to 0.) On T_2's side, the read of $y = 1$ updates T_2's `cur` view to $[x@0, y@t_y]$, and its acq view to $[x@t_x, y@t_y]$. The acquire fence then updates T_2's cur view to match its acq view, and hence the subsequent read of x must see the $x := 1$ write. If either the release or the acquire fence were missing, then T_2's cur view at the read of x would have been $[x@0, y@t_y]$, allowing it to read $x = 0$.

Interaction with Promises Promises (like every other message) now carry a view, and threads reading a promise are subject to the same constraints as if they were reading a normal message. In particular, after reading a promise and performing an acquire fence, a thread can only read messages with timestamp greater than or equal to the view carried in the promise message. In order to avoid cases where execution gets stuck, we must ensure that *some* message can be read for every location. Thus we require that the view attached to a promise message includes only timestamps of messages that exist in memory at the time the promise is made.

Going back to MP+fences, note that T_1 cannot promise $y := 1$ before performing $x := 1$. Indeed, because of the release fence, the view in the $y := 1$ message must include the message that will be produced for the $x := 1$ assignment, but at the beginning the only message for x in memory is the initial one (at timestamp 0). Hence, release fences effectively serve also as barriers for promises. We find it convenient to explicitly require this in our semantics: whenever a release fence is performed, the set of promises of the executing thread must be empty. This may seem restrictive, but note that the main reason for introducing promises was to allow read-write reorderings, as in the LB example of Section 3.2. If there is a release fence in between the read and write, then the reordering is no longer possible, and thus our motivation for promising the write is void.

Release/Acquire Accesses In addition to release and acquire fences, C/C++ offers a more fine-grained way of achieving synchronization, via *acquire reads* and *release writes*. Intuitively speaking, an acquire read is a relaxed read followed by an acquire fence, whereas a release write is a release fence followed by a relaxed write, with the restriction that these fences induce synchronization *only* on the location of the access. For example, in the following program,[1] only the second thread synchronizes with the first one.

$$\begin{array}{l||l||l} x := 1; & a := y_{\mathbf{acq}};\ /\!/\ 1 & c := z_{\mathbf{acq}};\ /\!/\ 1 \\ y_{\mathbf{rel}} := 1; & b := x;\ \ /\!/ \neq 0 & d := x;\ /\!/\ 0 \\ z := 1; & & \end{array}$$

[1] In this and in following code snippets, we annotate non-relaxed accesses with their access mode; all non-annotated accesses are relaxed.

Hence, b must get the value 1, while d may get 0.

To model these accesses, we treat the `rel` view of each thread not as a single view, but rather as one separate view per location, recording the thread's current view at the latest release fence or release write to that location. Thus, when a thread performs a release write to location x, we update its release view of x to match its *cur* view, while a release fence effects this update on the release views of *all* locations. Then, a write to x (either release or relaxed) will use the release view of x (newly updated, if it is a release write) to form the view of the write message, and an acquire read will incorporate the message's view into the reading thread's current view.

In the example above, at the end of T_1's execution, its thread view has $\mathtt{rel}(y) = [x@t_x, y@t_y, z@0]$, whereas $\mathtt{rel}(z) = [x@0, y@0, z@t_z]$. As a result, the $y_{\mathbf{acq}}$ read increases T_2's $\mathtt{cur}$ view to $[x@t_x, y@t_y, z@0]$, which forces it to then read $x = 1$, whereas the $z_{\mathbf{acq}}$ read increases T_3's $\mathtt{cur}$ view to $[x@0, y@0, z@t_z]$, which allows it to later read $x = 0$.

Release Sequences Using the per-location release views, we can straightforwardly handle C/C++-style *release sequences* (following the definition of release sequences given in [78]). In C/C++, an acquire read synchronizes with a release write w to x not only if it reads from w but also if it reads from a write in w's release sequence. The release sequence of w is inductively defined to include all the same-thread writes/updates to x after w, as well as all updates reading from an event in the release sequence of w. For example, in the following program, the $y_{\mathbf{acq}}$ synchronizes with the $y_{\mathbf{rel}} := 1$ because it reads from the FAA$(y, 1)$, which in turn reads from the $y := 2$.

$$\begin{array}{l||l||l}
x := 1; & & \\
y_{\mathbf{rel}} := 1; & \mathrm{FAA}(y, 1); & a := y_{\mathbf{acq}};\ /\!/\ 3 \\
y := 2; & & b := x;\ \ /\!/ \neq 0
\end{array}$$

Our operational semantics already handles the case of reading from a later write of the same thread, because the thread's release view for y is included in the message's view. To handle the updates that read from elements of the release sequence, we insist that the view of the write message of an update must incorporate the view of the read message of the update. Thus, in this example, the views of all the y messages contain $x@t_x$, and hence T_3 must read $x = 1$.

Promises Over Release/Acquire Accesses We finally point out another delicate issue related to the interaction between promises and release/acquire accesses. Consider the following variants of the LB example:

$$\begin{array}{l||l} a := x;\ \ /\!/ \neq 1 & \\ & x := y; \\ y_{\mathbf{rel}} := 1; & \end{array} \qquad \text{(LBr)} \qquad\qquad \begin{array}{l||l} a := x_{\mathbf{acq}};\ \ /\!/\ 1 & \\ & x := y; \\ y := 1; & \end{array} \qquad \text{(LBa)}$$

In the first variant (LBr), the promise of $y_{\mathbf{rel}} := 1$ should be forbidden for the same reason that a promise over a release fence is forbidden, and hence the specified behavior is disallowed. We note that this behavior is possible under the C/C++ model, but is not possible under the usual compilation of release writes to Power and ARM (using a `lwsync/dmb_sy` fence in the first thread).[2] More generally, our model forbids promises over release writes to the same location.

In the second variant (LBa), we allow the promise of $y := 1$ and thus the $a = 1$ outcome. The reason is that we want to enable optimizations that result in the elimination of an acquire read and thus remove the reordering constraints of the acquire. Consider, for example, the following program transformation:

$$\begin{array}{l||l} a := x;\ \ /\!/\ 2 & \\ y := 1; & \\ & x := y; \\ b := y_{\mathbf{acq}}; & \\ y := 2; & \end{array} \quad \leadsto \quad \begin{array}{l||l} y := 1; & \\ b := 1; & \\ & x := y; \\ y := 2; & \\ a := x;\ \ /\!/\ 2 & \end{array} \qquad \text{(LBa}'\text{)}$$

which may in effect reorder the $y := 2$ write before the $a := x$ read even though there is an acquire read in between (by first replacing $y_{\mathbf{acq}}$ with 1 and then reordering $a := x$ past both writes to y). Thus, our semantics has to allow promises over acquire actions. Note that there is no need to do so for release writes, because release writes cannot simply be eliminated in this way.

6.2 Sequentially Consistent (SC) Fences

We now extend the model with sequentially consistent (SC) fences, whose purpose is to allow the programmer to enforce strong ordering guarantees among memory accesses. In particular, full sequential consistency is restored if an SC fence is placed between every two shared memory accesses of a program.[3]

To handle SC fences, we extend our machine state with a *global* timemap $\mathcal{S}$, which records the latest messages written by any thread before an SC fence. When a thread T executes an SC fence, in addition to the effect of both an acquire and a release fence,

[2] Moreover, we observe that even the C/C++ model forbids this outcome, if we additionally make the read of y in the second thread into a consume read (which is supposed to be compiled exactly as a relaxed read, but preserving syntactic dependencies).

[3] In this regard, our semantics is stronger than the C/C++ model [13], which fails to validate this basic property, and follows Lahav *et. al.* [51, 53] instead.

T increases both its cur view and the global timemap to the maximum of the two. Consider the following variant of the SB example:

$$\begin{array}{l||l} x := 1; & y := 1; \\ \textbf{fence-sc}; & \textbf{fence-sc}; \\ a := y;\ /\!/\ 0 & b := x;\ \ /\!/\!\neq 0 \end{array} \qquad \text{(SB+fences)}$$

Here, the current views of the two threads just before their SC fences are $[x@t_x, y@0]$ and $[x@0, y@t_y]$, respectively, while the global view is $[x@0, y@0]$. If the fence of T_1 is executed first, it will update $\mathcal{S}$ to $[x@t_x, y@0]$. So, when the fence of T_2 is executed, both its cur view and $\mathcal{S}$ become $[x@t_x, y@t_y]$, from which point onwards T_2 *must* read $x = 1$.

6.3 "Plain" Non-Synchronizing Accesses

Both C/C++ and Java provide some form of *non-synchronizing* accesses, *i.e.*, accesses that are meant to be used only for non-racy data accesses (C++'s non-atomic accesses and Java's normal accesses). Such accesses can never achieve synchronization, even together with fences. Consequently, compilers are free to reorder non-synchronizing reads across acquire fences, and to reorder release fences across non-synchronizing writes. These non-synchronizing accesses, which we refer to as *plain* accesses, are easily supported in our model. The difference from relaxed accesses is simple: a plain read from a message m should not incorporate m.view into the thread's acq view; and a message m produced by a plain write should only carry the 0-view (*i.e.*, $\bot$ in the lattice of views). Moreover, plain writes can be promised even beyond a release fence or a release write to the same location.

Besides the reordering mentioned above, compilers can (and do) utilize further the assumption that some accesses are intended to be non-racy. Indeed, assuming two non-racy reads, a compiler may reorder them even if they are reading *the same* location. In a broader context, it may pave the way to further optimizations (*e.g.*, a compiler may prefer to unconditionally optimize $a := x; b := *p; c := x$ to $b := *p; a := x; c := a$, without the burden of analyzing whether the pointer p points to x or not). Since we followed C/C++'s assumption of full per-location coherence for our relaxed accesses, the reordering of two reads from the same location is unsound for them. Concretely, consider the following example:

$$\begin{array}{l||l} x := 1; & a := x;\ /\!/\ 2 \\ x := 2; & b := x;\ /\!/\ 1 \end{array} \quad \leadsto \quad \begin{array}{l||l} x := 1; & b := x;\ /\!/\ 1 \\ x := 2; & a := x;\ /\!/\ 2 \end{array}$$

The target program obviously allows the specified behavior, while the source does not. Fortunately, it is not hard to adapt our plain accesses to provide only partial per-location coherence (in C18/C++17 terms, dropping "coherence-RR" for plain accesses), consequently allowing this reordering. The idea is to extend the notion of a view V—both message views R and the three component views of a thread ($\mathtt{cur}, \mathtt{acq}, \mathtt{rel}$)—from being a single timemap to a pair of timemaps: a "normal" one ($V.\mathtt{rlx}$) as before, and one for plain accesses ($V.\mathtt{pln}$). The $V.\mathtt{pln}$ timemap is generally smaller than the normal timemap ($V.\mathtt{pln} \leq V.\mathtt{rlx}$), and restricts the possible timestamps available to plain reads. A plain read from a message m with location x and time t only consults this new timemap, checking that $\mathtt{cur.pln}(x) \leq t$, and only updates $\mathtt{cur.rlx}(x)$ to include t. A plain write, on the other hand, cannot pick a timestamp smaller than $\mathtt{cur.rlx}(x)$ (since we do maintain the other coherence properties besides "coherence-RR").

Importantly, we do not exploit "catch-fire" semantics (à la C/C++) to accommodate our plain accesses, but rather give a well-defined semantics to arbitrary racy programs. In addition, we note that it is easy to decouple the two weaknesses of plain accesses compared to relaxed ones, by introducing a middle access mode that allows synchronization (together with release and acquire fences) but supports only partial per-location coherence.

Remark 1. Our model handles only hardware-atomic memory accesses. To handle non-atomic reads/writes, such as Java double and C struct accesses, our semantics could be extended by introducing "garbage values" (LLVM-style undefined values [3]) as in [21].

6.4 System Calls

For the purpose of defining the behaviors of programs (as needed to prove soundness of transformations), we augment our language and semantics with system calls labeled with "SysCall(v)". These are operations that are visible to an external observer (*e.g.*, printing statements). For simplicity, we assume that these take one value (input or output), and more importantly, that they do not access the memory, and serve as the strongest barrier for reordering. Thus, we simply model system calls as SC fences.

6.5 Modifying Existing Promises

So far, our model does not allow promises, once made, to be changed. However, our full model does allow two forms of promise adjustment, both of which are defined in such a way that threads that have already read from the promised message are unaffected.

Split The first form of promise adjustment is *splitting*. Consider the following example:

$$\left.\begin{array}{l} a := x; \;\; /\!/\, 2 \\ \textbf{if } a = 2 \textbf{ then } \mathrm{FAA}(y,1); \mathrm{FAA}(y,1); \;\; \textbf{else } \mathrm{FAA}(y,2); \end{array} \right\Vert \begin{array}{l} x := y; \end{array}$$

We find it natural to allow the specified behavior, as it can be obtained by benign compiler optimizations: first $\mathrm{FAA}(y,1); \mathrm{FAA}(y,1)$ can be merged to $\mathrm{FAA}(y,2)$, and then the whole if-then-else statement can be replaced by $\mathrm{FAA}(y,2)$. Nevertheless, the model described so far forbids this behavior. Indeed, clearly, an execution obtaining this behavior must start with T_1 promising $\langle y : 2@(f,t]\rangle$. Since this promise must be certified under an arbitrary future memory, T_1 must pick $f = 0$ (or else, it cannot fulfill its promise for a memory that includes, say, $\langle y : 42@(0,5]\rangle$). Then, T_2 can read the promise and add a message of the form $\langle x : 2@(\_, t_x]\rangle$ to the memory. Now, T_1 would like to read this message. However, if it does so, it will not be able to fulfill its promise $\langle y : 2@(0,t]\rangle$, simply because there is no available timestamp interval in which it can put the first $y = 1$ message. To solve this, we allow threads to split their own promises in two pieces, keeping the original promise with the same $m.\mathtt{to}$ value. For the example above, T_1 could proceed by splitting its promise $\langle y : 2@(0,t]\rangle$ into $\langle y : 1@(0,t/2]\rangle$ and $\langle y : 2@(t/2,t]\rangle$, reading the message $\langle x : 2@(\_, t_x]\rangle$ and fulfilling both promises.

Lower The second form of promise adjustment is *lowering* of the promised message's view. Note that by promising a message carrying a high view, a thread places *more* restrictions on the readers of that promise. Thus, changing the view of a promise m to a view $R' \leq m.\mathtt{view}$ can never cause any harm. Technically, including this option simplifies our simulation arguments used to prove the soundness of program transformations, by allowing us to have a simpler simulation relation between the source and target memories. More generally speaking, it allows us to prove and use the following natural property: if all the views included in some machine state **MS** (in its memory's messages and its threads' views) are less than or equal to all views in another machine state **MS**$'$, then every behavior of **MS**$'$ is also a behavior of **MS**.

6.6 Formal Model

Finally, we formally present our full model, combining and making precise all the ideas outlined above. The model employs three modes for memory accesses, naturally ordered as follows:

$$\mathtt{pln} \sqsubset \mathtt{rlx} \sqsubset \mathtt{ra}$$

We use o as a metavariable for access mode. The programming language is modeled by a transition system whose transition labels (see Section 4.2) are: "Silent" for local transitions; $\mathtt{R}(o, x, v)$ for reads; $\mathtt{W}(o, x, v)$ for writes; $\mathtt{U}(o_{\mathrm{r}}, o_{\mathrm{w}}, x, v_{\mathrm{r}}, v_{\mathrm{w}})$ for updates; $\mathtt{F}_{\mathtt{acq}}, \mathtt{F}_{\mathtt{rel}}, \mathtt{F}_{\mathtt{sc}}$ for fences; and SysCall(v) for system calls. Note that updates have two access modes, one for the read and one for the write; and that only fences may have the $\mathtt{sc}$ mode.

View A *view* is a pair $V = \langle T_{\mathtt{pln}}, T_{\mathtt{rlx}} \rangle$ of timemaps (see Section 4.2) satisfying $T_{\mathtt{pln}} \leq T_{\mathtt{rlx}}$. We denote by $V.\mathtt{pln}$ and $V.\mathtt{rlx}$ the components of V. View denotes the set of all views.

Messages A *message* m is a tuple $\langle x : v@(f, t], R \rangle$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $f, t \in \mathsf{Time}$, and $R \in \mathsf{View}$, such that $f < t$ or $f = t = 0$, and $R.\mathtt{rlx}(x) = t$ or $R = \bot$. We denote by $m.\mathtt{loc}$, $m.\mathtt{val}$, $m.\mathtt{from}$, $m.\mathtt{to}$, and $m.\mathtt{view}$ the components of m.

Memory A *memory* is a (nonempty) pairwise disjoint finite set of messages (see Section 5 for def. of disjointness). A memory M supports the following insertions of a message $m = \langle x : v@(f, t], R \rangle$:

- The *additive insertion*, denoted by $M \overset{\mathrm{A}}{\hookleftarrow} m$, is only defined if $\{m\} \# M$, in which case it is given by $\{m\} \cup M$.
- The *splitting insertion*, denoted by $M \overset{\mathrm{S}}{\hookleftarrow} m$, is only defined if there exists $m' = \langle x : v'@(f, t'], R' \rangle$ with $t < t'$ in M, in which case it is given by $M \setminus \{m'\} \cup \{m, \langle x : v'@(t, t'], R' \rangle\}$.
- The *lowering insertion*, denoted by $M \overset{\mathrm{U}}{\hookleftarrow} m$, is only defined if there exists $m' = \langle x : v@(f, t], R' \rangle$ with $R \leq R'$ in M, in which case it is given by $M \setminus \{m'\} \cup \{m\}$.

We write $M(x)$ for the sub-memory $\{\, m \in M \mid m.\mathtt{loc} = x \,\}$.

Closed Memory Given a timemap T and a memory M, we write $T \in M$ if, for every $x \in \mathsf{Loc}$, we have $T(x) = m.\mathtt{to}$ for some $m \in M$ with $m.\mathtt{loc} = x$. For a view V, we write $V \in M$ if $T \in M$ for each component timemap T of V. A memory M is *closed* if $m.\mathtt{view} \in M$ for every $m \in M$.

Future Memory For memories M, M', we write $M \to M'$ if $M' \in \{M \overset{\mathrm{A}}{\hookleftarrow} m, M \overset{\mathrm{S}}{\hookleftarrow} m, M \overset{\mathrm{U}}{\hookleftarrow} m\}$ for some message m, and M' is closed. We say that M' is a *future memory* of M w.r.t. a memory P, if $P \subseteq M'$ and $M \to^{*} M'$.

(MEMORY: NEW)

$$\frac{}{\langle P, M\rangle \xrightarrow{m} \langle P, M \overset{\mathrm{A}}{\hookleftarrow} m\rangle}$$

(MEMORY: FULFILL)

$$\frac{\hookleftarrow \in \{\overset{\mathrm{S}}{\hookleftarrow}, \overset{\mathrm{U}}{\hookleftarrow}\} \qquad P' = P \hookleftarrow m \qquad M' = M \hookleftarrow m}{\langle P, M\rangle \xrightarrow{m} \langle P' \setminus \{m\}, M'\rangle}$$

(READ-HELPER)

$$\frac{\begin{array}{c} o = \texttt{pln} \implies cur.\texttt{pln}(x) \le t \\ o \in \{\texttt{rlx}, \texttt{ra}\} \implies cur.\texttt{rlx}(x) \le t \\ cur' = cur \sqcup V \sqcup (o \sqsupseteq \texttt{ra}\ ?\, R) \\ acq' = acq \sqcup V \sqcup (o \sqsupseteq \texttt{rlx}\, ?\, R) \\ \textit{where } V = [\texttt{pln} : (o \sqsupseteq \texttt{rlx}\, ?\, \{x@t\}), \texttt{rlx} : \{x@t\}] \end{array}}{\langle cur, acq, rel\rangle \xrightarrow{\mathsf{R}:o,x,t,R} \langle cur', acq', rel\rangle}$$

(WRITE-HELPER)

$$\frac{\begin{array}{c} cur.\texttt{rlx}(x) < t \\ cur' = cur \sqcup V \qquad acq' = acq \sqcup cur' \\ rel' = rel[x \mapsto rel(x) \sqcup V \sqcup (o \sqsupseteq \texttt{ra}\, ?\, cur')] \\ R_{\mathrm{w}} = (o \sqsupseteq \texttt{rlx}\, ?\, (rel'(x) \sqcup R_{\mathrm{r}})) \\ \textit{where } V = [\texttt{pln} : \{x@t\}, \texttt{rlx} : \{x@t\}] \end{array}}{\langle cur, acq, rel\rangle \xrightarrow{\mathsf{W}:o,x,t,R_{\mathrm{r}},R_{\mathrm{w}}} \langle cur', acq', rel'\rangle}$$

(SC-FENCE-HELPER)

$$\frac{\begin{array}{c} \mathcal{S}' = acq.\texttt{rlx} \sqcup \mathcal{S} \\ cur' = acq' = \langle \mathcal{S}', \mathcal{S}'\rangle \\ rel' = \lambda\_.\langle \mathcal{S}', \mathcal{S}'\rangle \end{array}}{\begin{array}{c} \langle\langle cur, acq, rel\rangle, \mathcal{S}\rangle \xrightarrow{\mathsf{F_{sc}}} \\ \langle\langle cur', acq', rel'\rangle, \mathcal{S}'\rangle \end{array}}$$

(READ)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathsf{R}(o,x,v)} \sigma' \\ \langle x : v@(\_, t], R\rangle \in M \\ \mathcal{V} \xrightarrow{\mathsf{R}:o,x,t,R} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}, M\rangle}$$

(WRITE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathsf{W}(o,x,v)} \sigma' \\ o = \texttt{ra} \implies \forall m' \in P(x).\ m'.\texttt{view} = \bot \\ m = \langle x : v@(\_, t], R\rangle \\ \langle P, M\rangle \xrightarrow{m} \langle P', M'\rangle \\ \mathcal{V} \xrightarrow{\mathsf{W}:o,x,t,\bot,R} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, \mathcal{S}, M'\rangle}$$

(UPDATE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathsf{U}(o_{\mathrm{r}},o_{\mathrm{w}},x,v_{\mathrm{r}},v_{\mathrm{w}})} \sigma' \\ o_{\mathrm{w}} = \texttt{ra} \implies \forall m' \in P(x).\ m'.\texttt{view} = \bot \\ \langle x : v_{\mathrm{r}} @(\_, t_{\mathrm{r}}], R_{\mathrm{r}}\rangle \in M \\ m_{\mathrm{w}} = \langle x : v_{\mathrm{w}}@(t_{\mathrm{r}}, t_{\mathrm{w}}], R_{\mathrm{w}}\rangle \\ \langle P, M\rangle \xrightarrow{m_{\mathrm{w}}} \langle P', M'\rangle \\ \mathcal{V} \xrightarrow{\mathsf{R}:o_{\mathrm{r}},x,t_{\mathrm{r}},R_{\mathrm{r}}} \xrightarrow{\mathsf{W}:o_{\mathrm{w}},x,t_{\mathrm{w}},R_{\mathrm{r}},R_{\mathrm{w}}} \mathcal{V}' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}', P'\rangle, \mathcal{S}, M'\rangle}$$

(ACQ-FENCE)

$$\frac{\sigma \xrightarrow{\mathsf{F_{acq}}} \sigma' \qquad cur' = acq}{\begin{array}{c} \langle\langle \sigma, \langle cur, acq, rel\rangle, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \langle\langle cur', acq, rel\rangle, P\rangle, \mathcal{S}, M\rangle\rangle \end{array}}$$

(REL-FENCE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathsf{F_{rel}}} \sigma' \quad rel' = \lambda\_.cur \\ \forall m \in P.\ m.\texttt{view} = \bot \end{array}}{\begin{array}{c} \langle\langle \sigma, \langle cur, acq, rel\rangle, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \langle\langle cur, acq, rel'\rangle, P\rangle, \mathcal{S}, M\rangle\rangle \end{array}}$$

(SC-FENCE)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathsf{F_{sc}}} \sigma' \\ \langle \mathcal{V}, \mathcal{S}\rangle \xrightarrow{\mathsf{F_{sc}}} \langle \mathcal{V}', \mathcal{S}'\rangle \\ \forall m \in P.\ m.\texttt{view} = \bot \end{array}}{\begin{array}{c} \langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \\ \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}', M\rangle \end{array}}$$

(SYSTEM CALL)

$$\frac{\begin{array}{c} \sigma \xrightarrow{\mathrm{SysCall}(v)} \sigma' \\ \langle \mathcal{V}, \mathcal{S}\rangle \xrightarrow{\mathsf{F_{sc}}} \langle \mathcal{V}', \mathcal{S}'\rangle \\ \forall m \in P.\ m.\texttt{view} = \bot \end{array}}{\begin{array}{c} \langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \xrightarrow{\mathrm{SysCall}(v)} \\ \langle\langle \sigma', \mathcal{V}', P\rangle, \mathcal{S}', M\rangle \end{array}}$$

(SILENT)

$$\frac{\sigma \xrightarrow{\mathrm{Silent}} \sigma'}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma', \mathcal{V}, P\rangle, \mathcal{S}, M\rangle}$$

(PROMISE)

$$\frac{\begin{array}{c} \hookleftarrow \in \{\overset{\mathrm{A}}{\hookleftarrow}, \overset{\mathrm{S}}{\hookleftarrow}, \overset{\mathrm{U}}{\hookleftarrow}\} \quad P' = P \hookleftarrow m \\ M' = M \hookleftarrow m \quad m.\texttt{view} \in M' \end{array}}{\langle\langle \sigma, \mathcal{V}, P\rangle, \mathcal{S}, M\rangle \to \langle\langle \sigma, \mathcal{V}, P'\rangle, \mathcal{S}, M'\rangle}$$

(MACHINE STEP)

$$\frac{\begin{array}{c} \langle \mathcal{TS}(i), \mathcal{S}, M\rangle \to^{*} \langle TS', \mathcal{S}', M'\rangle \\ \langle TS', \mathcal{S}', M'\rangle \xrightarrow{e} \langle TS'', \mathcal{S}'', M''\rangle \\ \langle TS'', \mathcal{S}'', M''\rangle \text{ is consistent} \end{array}}{\langle \mathcal{TS}, \mathcal{S}, M\rangle \xrightarrow{e} \langle \mathcal{TS}[i \mapsto TS''], \mathcal{S}'', M''\rangle}$$

Figure II.3 Full operational semantics

Threads A *thread view* is a triple $\mathcal{V} = \langle cur, acq, rel \rangle$, where $cur, acq \in \mathsf{View}$ and $rel \in \mathsf{Loc} \to \mathsf{View}$ satisfying $rel(x) \leq cur \leq acq$ for all $x \in \mathsf{Loc}$. We denote by $\mathcal{V}.\mathtt{cur}$, $\mathcal{V}.\mathtt{acq}$, and $\mathcal{V}.\mathtt{rel}$ the components of $\mathcal{V}$. A *thread state* is a triple $TS = \langle \sigma, \mathcal{V}, P \rangle$ defined just as in Section 4.2 except with a thread view $\mathcal{V}$ instead of a single timemap (σ is a local state and P is a memory). We denote by $TS.\mathtt{st}$, $TS.\mathtt{view}$, and $TS.\mathtt{prm}$ the components of TS.

Thread Configuration Steps A *thread configuration* is a triple $\langle TS, \mathcal{S}, M \rangle$, where TS is a thread state, $\mathcal{S}$ is a timemap (the global SC timemap), and M is a memory.

Figure II.3 presents the full list of thread configuration steps. To avoid repetition, we use the additional rules READ-HELPER, WRITE-HELPER, and SC-FENCE-HELPER. These employ several helpful notations: $\bot$ and $\sqcup$ denote the natural bottom elements and join operations for timemaps and for views (pointwise extensions of the initial timestamp 0 and the $\sqcup$—*i.e.*, max—operation on timestamps); $\{x@t\}$ denotes the timemap assigning t to x and 0 to other locations; and $(cond\,?\,X)$ is defined to be X if $cond$ holds, and $\bot$ otherwise.

The write and the update steps cover two cases: a fresh write (MEMORY:NEW) and a fulfillment of an outstanding promise (MEMORY:FULFILL). The latter allows to split the promise or lower its view before its fulfillment (note that when $m \in P \subseteq M$, we have $P = P \overset{\cup}{\hookleftarrow} m$ and $M = M \overset{\cup}{\hookleftarrow} m$ by def. of $\overset{\cup}{\hookleftarrow}$).

Consistency A thread configuration $\langle TS, \mathcal{S}, M \rangle$ is called *consistent* if for every future memory M_{future} of M w.r.t. $TS.\mathtt{prm}$ and every timemap $\mathcal{S}_{\text{future}}$ with $\mathcal{S} \leq \mathcal{S}_{\text{future}} \in M_{\text{future}}$, there exist TS', $\mathcal{S}'$, M' such that:

$$\langle TS, \mathcal{S}_{\text{future}}, M_{\text{future}} \rangle \to^{*} \langle TS', \mathcal{S}', M' \rangle \;\wedge\; TS'.\mathtt{prm} = \varnothing$$

Machine and Behaviors A *machine state* is a triple $\mathbf{MS} = \langle \mathcal{TS}, \mathcal{S}, M \rangle$ consisting of a function $\mathcal{TS}$ assigning a thread state to every thread, an SC timemap $\mathcal{S}$, and a memory M. The initial state $\mathbf{MS}^0$ (for a given program) consists of the function $\mathcal{TS}^0$ mapping each thread i to its initial state σ_i^0, the zero thread view (all timestamps in all timemaps are 0), and an empty set of promises; the zero timemap $\mathcal{S}^0$; and the initial memory M^0 consisting of one message $\langle x : 0@(0, 0], \bot \rangle$ for each location x. The machine step is defined by the last rule in Figure II.3. The variable e in the final thread configuration step can either be a usual step (e is empty), or denote a system call ($e = \mathrm{SysCall}(v)$).

To define the set of behaviors of a program $\mathcal{P}$ (namely, what is externally observable during $\mathcal{P}$'s executions), we use the system calls that $\mathcal{P}$'s executions perform. More precisely, every execution induces a sequence of system calls (each includes a specific value for input/output), and the set of behaviors of $\mathcal{P}$ is taken to be the set of all system call sequences induced by executions of $\mathcal{P}$.

Promise-Free Machine In several of our results below, we make use of the fragment of our model obtained by revoking the ability to make promises (*i.e.*, omitting the PROMISE rule). We call this the *promise-free machine*.

7 Results

In this section, we outline a number of important results we have proven to hold of our "promising" model. All the results of this section **are fully validated in Coq** except for Theorems 2 and 3, for which we provide hand-written proofs in Section 8. The Coq development is available at [1].

7.1 Compiler Transformations

A transformation $\mathcal{P}_{\text{src}} \rightsquigarrow \mathcal{P}_{\text{tgt}}$ is *sound* if it does not introduce new behaviors under any (parallel and sequential) context, that is, for every context $\mathcal{C}$, every behavior of $\mathcal{C}[\mathcal{P}_{\text{tgt}}]$ is a behavior of $\mathcal{C}[\mathcal{P}_{\text{src}}]$.

Next, we list the program transformations proven to be sound in our model. To streamline the presentation, we refer to transformations on the *semantic* level, as if they are applied to *actions*, namely fences and (valueless) memory accesses. Thus, we presuppose adequate syntactic manipulations on the program level that implement these semantic transformations. For example, a syntactic transformation implementing $\mathsf{R}^x_{\mathtt{rlx}}; \mathsf{R}^y_{\mathtt{rlx}} \rightsquigarrow \mathsf{R}^y_{\mathtt{rlx}}; \mathsf{R}^x_{\mathtt{rlx}}$ is a reordering $a := x; b := y \rightsquigarrow b := y; a := x$ on the program code (assuming $a \neq b$); while a merge of a write and an update correspond, *e.g.*, to a transformation of the form $x := a; \text{FAA}(x, 1) \rightsquigarrow x := a + 1$. Nevertheless, our formal development proves soundness of transformations on the purely *syntactic* level, assuming a simple programming language with memory operations, conditionals, and loops.

Trace-Preserving Transformations Transformations that do not change the set of traces of actions in a given thread are clearly sound. For example, $y := a + 1 - a \rightsquigarrow y := 1$ is a sound transformation (recall that a denotes a local register; see Section 3:LBfd). Indeed, this is the crucial property that distinguishes a memory model for a higher-level language from a hardware memory model.

Strengthening A simple transformation that is sound in our model is strengthening of access modes. A read/write action $\mathtt{X}_o$ can be transformed to $\mathtt{X}_{o'}$ provided that $o \sqsubseteq o'$. Similarly, it is sound to replace $\mathtt{U}_{o_\mathrm{r},o_\mathrm{w}}$ by $\mathtt{U}_{o'_\mathrm{r},o'_\mathrm{w}}$ provided that $o_\mathrm{r} \sqsubseteq o'_\mathrm{r}$ and $o_\mathrm{w} \sqsubseteq o'_\mathrm{w}$, or to strengthen $\mathtt{F}_\mathtt{rel}$ or $\mathtt{F}_\mathtt{acq}$ to $\mathtt{F}_\mathtt{sc}$.

Reordering Next we consider transformations of the form $\mathtt{X};\mathtt{Y} \rightsquigarrow \mathtt{Y};\mathtt{X}$, and specify the set of *reorderable* pairs, that is the set of pairs $\mathtt{X};\mathtt{Y}$ for which we proved this reordering transformation to be sound in our model. First, the following pairs are reorderable (where x and y denote distinct locations):

- $\mathtt{W}^x;\mathtt{R}^y$
- $\mathtt{R}^x_{\sqsubseteq\mathtt{rlx}};\mathtt{R}^y$ and $\mathtt{R}^x_\mathtt{pln};\mathtt{R}^x_\mathtt{pln}$
- $\mathtt{F}_\mathtt{rel};\mathtt{W}_{\neq\mathtt{rlx}}$
- $\mathtt{W}^x;\mathtt{W}^y_{\sqsubseteq\mathtt{rlx}}$
- $\mathtt{R}^x_{\sqsubseteq\mathtt{rlx}};\mathtt{W}^y_{\sqsubseteq\mathtt{rlx}}$
- $\mathtt{F}_\mathtt{rel};\mathtt{R}$
- $\mathtt{W};\mathtt{F}_\mathtt{acq}$
- $\mathtt{R}_{\neq\mathtt{rlx}};\mathtt{F}_\mathtt{acq}$
- $\mathtt{F}_\mathtt{rel};\mathtt{F}_\mathtt{acq}$

In addition, for the purpose of specifying reorderable pairs, an update is just a combination of a read and a write. Thus, $\mathtt{X};\mathtt{U}_{o_\mathrm{r},o_\mathrm{w}}$ is reorderable if both $\mathtt{X};\mathtt{R}_{o_\mathrm{r}}$ and $\mathtt{X};\mathtt{W}_{o_\mathrm{w}}$ are reorderable, and symmetrically $\mathtt{U}_{o_\mathrm{r},o_\mathrm{w}};\mathtt{X}$ is reorderable if both $\mathtt{R}_{o_\mathrm{r}};\mathtt{X}$ and $\mathtt{W}_{o_\mathrm{w}};\mathtt{X}$ are reorderable. In particular, a pair $\mathtt{U}^x_{o^x_\mathrm{r},o^x_\mathrm{w}};\mathtt{U}^y_{o^y_\mathrm{r},o^y_\mathrm{w}}$ is reorderable if $x \neq y$, $o^x_\mathrm{r} \sqsubseteq \mathtt{rlx}$, $o^y_\mathrm{w} \sqsubseteq \mathtt{rlx}$.

The set of reorderable pairs in our model contains all pairs that are intended to be reorderable in the C/C++ and Java memory models, including in particular all "roach-motel reorderings" [78, 72].

Merging These are transformations that completely eliminate an action. Clearly, the two actions in *mergeable* pairs (pairs for which we proved the merge to be sound in our model) should access the same location. The following three kinds of pairs are mergeable:

R-after-R$\mathtt{R}_o;\mathtt{R}_o$ W-after-W$\mathtt{W}_o;\mathtt{W}_o$ R-after-W: $\mathtt{W};\mathtt{R}$

Using the strengthening transformation, the access modes here can be read as upper bounds (*e.g.*, $\mathtt{R}_\mathtt{ra};\mathtt{R}_\mathtt{rlx}$ can be first strengthened to $\mathtt{R}_\mathtt{ra};\mathtt{R}_\mathtt{ra}$ and then merged). Note that the R-after-W merge allows even to eliminate a redundant acquire read after a plain/relaxed write (as in Example LBa′ in Section 6.1).

In addition, the following pairs involving updates are mergeable:

R-after-U: $\mathtt{U}_{\mathtt{rlx},o};\mathtt{R}_\mathtt{rlx}$, and $\mathtt{U}_{\mathtt{ra},o};\mathtt{R}_\mathtt{ra}$ U-after-W: $\mathtt{W}_o;\mathtt{U}_{o_\mathrm{r},o}$

U-after-U: $\mathtt{U}_{o_1,o};\mathtt{U}_{o_2,o}$, provided that $\mathtt{U}_{o_1,o};\mathtt{R}_{o_2}$ is mergeable

Note that read-after-update does not allow the read to be an acquire read unless the update includes an acquire read (unlike read-after-write elimination). This is due to release sequences: eliminating an acquire read after a relaxed update may remove the synchronization due to a release sequence ending in this update.

Finally, two fences of the same type can obviously be merged.

The set of mergeable pairs in our model contains all pairs intended to be mergeable in the C/C++ and Java models [78, 72]. In particular, we support `R`-after-`W` merging, which is the effect of local satisfaction of reads in hardware like TSO, Power, and ARM.

Introducing and Eliminating Unused Reads Introduction of irrelevant read accesses is sound in our model, unlike in the Java memory model [72]. Eliminating plain read accesses whose read values are never used in the program is also sound in our model. In contrast, eliminating relaxed or acquire reads is not generally sound because it may remove synchronization.

Proof Technique Our proof of these results employs the well-known approach of simulation relations between the target and the source programs. We prove the adequacy of simulation up-to context, or in other words, we can prove simulations between code fragments and compose them. Our definitions are fairly standard, except that they ensure thread-locality, thus allowing us to define a simulation relation on thread configurations, which (as we prove) can be composed into a simulation relation on full machine states. The thread-locality is crucially used in proving the soundness of transformations because they are performed not for a full machine but for a single thread. Our thread-local simulation relation is based on novel ideas for reasoning about the shared resources (*i.e.*, the SC timemap and the memory) and consistency. For more details, we refer the reader to Section 8.1.

7.2 Compilation to TSO

Like C18/C++17, our model can be efficiently compiled to x86-TSO. Since this architecture provides relatively strong guarantees, every memory access can be compiled to a primitive hardware instruction. Moreover, release/acquire fences are ignored during compilation, and SC fences are mapped to an `MFENCE` instruction. Correctness of this mapping follows from a recent result by Lahav and Vafeiadis [52], which shows that all weak behaviors of TSO are explained by store-load reordering and merging. Accordingly, it reduces the correctness proof of compilation to TSO to: (i) supporting write-read reordering and write-read merge; and (ii) a correctness proof of compilation to

SC. Since we proved the soundness of write-read reordering and merge (regardless of the access modes of the two events), and since clearly our model is weaker than SC, we immediately derive the correctness of compilation to TSO.

7.3 DRF Theorems

We proceed with an explanation of our DRF theorems. These theorems provide ways of restricting attention to better-behaved subsets of the model assuming certain conditions on programs.

Evidently, the most complicated part of our semantics is the promises. Without promises, our model amounts to a usual operational model, where thread steps only arise because of program instructions. Hence, our first DRF result (and the one that is by far the most challenging to prove) identifies a set of programs for which promises cannot introduce additional behaviors. Specifically, we show that this holds for programs in which all racy accesses are release/acquire, assuming a promise-free semantics. Crucially, as usual in DRF guarantees, the races are considered under the stronger semantics (promise-free), not the full model, thus allowing programmers to adhere to this programming discipline while being completely ignorant of the weak semantics (promises).

More precisely, we say that a machine state **MS** is *o-race-free*, if whenever two different threads may take a (non-promise) step accessing the same location, then both accesses are reads or both have access mode strictly stronger than o.

Theorem 1 (Promise-Free DRF)**.** Let $\Longrightarrow$ denote the steps of the promise-free machine (see end of 6.6). Suppose that every machine state that is $\Longrightarrow$-reachable from the initial state of a program $\mathcal{P}$ is `rlx`-race-free. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\Longrightarrow$-machine.

Putting promises aside, a counter-intuitive part of weak memory models are the relaxed accesses, which allow threads to observe writes without observing previous writes to other locations. Removing `pln`/`rlx` accesses, namely keeping only `ra`, substantially simplifies our machine (in particular, its thread views would consist of just one view, the `cur` one). Accordingly, our second DRF result strengthens Theorem 1 and states that it suffices to show that there are only races on `ra` accesses *under release/acquire semantics* to conclude that a program has only release/acquire behaviors.

Theorem 2 (DRF-RA)**.** Let $\overset{\texttt{ra}}{\Longrightarrow}$ be identical to $\Longrightarrow$ in Theorem 1, except for interpreting `rlx` and `pln` accesses in program transitions as if they are all `ra`-accesses. Suppose that every machine state that is $\overset{\mathsf{ra}}{\Longrightarrow}$-reachable from the initial state of a program $\mathcal{P}$ is $\mathtt{rlx}$-race-free. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\overset{\mathsf{ra}}{\Longrightarrow}$-machine.

To state a more standard DRF theorem, we assume programs are *well-locked*: (1) locations are partitioned into *normal* and *lock* locations, and (2) lock locations are accessed only by matching pairs of the following lock/unlock operations:

$$\begin{aligned} lock(l): &\quad \textbf{while } !\mathrm{CAS}(l, 0, 1, \textbf{acqrel}) \textbf{ do skip};\\ unlock(l): &\quad l_{\textbf{rel}} := 0; \end{aligned}$$

The theorem forbids any weak behavior in programs that, under SC semantics, race only on lock locations. For the SC semantics, we consider "an interleaving machine", where reads read from the latest write to the appropriate location (regardless of the access modes).

Theorem 3 (DRF-LOCK)**.** Let $\overset{\mathsf{sc}}{\Longrightarrow}$ denote the steps of the interleaving machine. Suppose that every machine state that is $\overset{\mathsf{sc}}{\Longrightarrow}$-reachable from the initial state of a well-locked program $\mathcal{P}$ is race-free on normal locations. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\overset{\mathsf{sc}}{\Longrightarrow}$-machine.

Necessity of Re-certification We observe that promise re-certification is necessary for Theorems 1 and 2 to hold, as shown in the following example:

```
                 ‖                          ‖ $a := w_{\mathbf{acq}}$;
                 ‖                          ‖ if ($a == 1$) {
                 ‖ $a := y_{\mathbf{acq}}$;             ‖   $z := 1$;
                 ‖ if ($a == 1$) {           ‖ else {
                 ‖   $b := z$;                ‖   $y_{\mathbf{rel}} := 1$;
$w_{\mathbf{rel}} := 1$;  ‖   if ($b == 1$) {         ‖ }
                 ‖     $x := 1$;              ‖ $b := x$; // 1
                 ‖   }                      ‖ if ($b == 1$) {
                 ‖ }                        ‖   $z := 1$;
                 ‖                          ‖ }
```

This behavior contradicts both of Theorems 1 and 2, and yet it is allowed in the absence of promise re-certification. Consider the following execution. First, Thread 1 writes $w = 1$ and Thread 3 promises to write $z = 1$. Now Thread 3 reads $w = 0$, after which it can no longer certify to write the promised value $z = 1$. Then Thread 3 writes $y = 1$, Thread 2 reads $y = 1$ and $z = 1$, Thread 2 writes $x = 1$, Thread 3 reads $x = 1$, and then Thread 3 fulfills the promise to write $z = 1$.

7.4 An Invariant-Based Program Logic

Besides the DRF guarantees, to demonstrate that our model does not suffer from the disastrous consequences of OOTA, we prove soundness of a very simple program logic for concurrent programs with respect to our model. In particular, it can be trivially used to show that LBd must return $a = 0$, and more generally, that programs cannot read values they never wrote. Note that even this basic logic is unsound for C/C++'s relaxed accesses (whereas it is sound in our model even if all accesses are plain).

We take a *program proof* to be a tuple $\langle J, S_1, S_2, \dots \rangle$, where J is a global invariant over the shared variables and each $S_i \subseteq \mathtt{State}_i$ is a set of local states (intuitively describing the reachable states of thread i) such that the following conditions hold:

- $\sigma_i^0 \in S_i$ and $\bigwedge_{x \in \mathtt{Loc}} x = 0 \vdash J$.
- If $\sigma_i \xrightarrow{\mathtt{R}(o,x,v)} \sigma_i'$ then $\sigma_i \in S_i \wedge J \wedge x = v \vdash \sigma_i' \in S_i$.
- If $\sigma_i \xrightarrow{\mathtt{W}(o,x,v)} \sigma_i'$ then $\sigma_i \in S_i \wedge J \vdash \sigma_i' \in S_i \wedge J[v/x]$.
- If $\sigma_i \xrightarrow{\mathtt{U}(o_{\mathrm{r}},o_{\mathrm{w}},x,v_{\mathrm{r}},v_{\mathrm{w}})} \sigma_i'$ then $\sigma_i \in S_i \wedge J \wedge x = v_{\mathrm{r}} \vdash \sigma_i' \in S_i \wedge J[v_{\mathrm{w}}/x]$.
- For $e \in \{\mathtt{F}_{\mathtt{acq}}, \mathtt{F}_{\mathtt{rel}}, \mathtt{F}_{\mathtt{sc}}, \mathrm{Silent}, \mathrm{SysCall}(v)\}$, if $\sigma_i \xrightarrow{e} \sigma_i'$ then $\sigma_i \in S_i \vdash \sigma_i' \in S_i$.

Figure II.4 provides an illustration of a program proof showing that LBd does not exhibit weak behaviors.

Now, given a program proof for a program $\mathcal{P}$, we can show that all the reachable states $\mathbf{MS}$ from the initial state $\mathbf{MS}^0$ of $\mathcal{P}$ satisfy the global invariant J:

Theorem 4 (Soundness). Let $\langle J, S_1, S_2, \dots \rangle$ be a program proof, and let $\mathbf{MS} = \langle TS, \mathcal{S}, M \rangle$ such that $\mathbf{MS}^0 \rightarrow^* \mathbf{MS}$. Then, $TS(i).\mathtt{st} \in S_i$ for every thread i, and $\bigwedge_{x \in \mathtt{Loc}} x = f(x).\mathtt{val} \vdash J$ for every function f that assigns to every location x a message $m \in M$ such that $m.\mathtt{loc} = x$.

Our Coq proof of this theorem is simple: it holds trivially for promise-free executions, and extends easily to promise steps, since every promise step has a promise-free certification.

$$\begin{array}{l||l}
\{J\} & \\
a := x; & \{J\} \\
\{J \land (a = 0)\} & x := y; \\
y := a; & \{J\} \\
\{J\} &
\end{array}
\qquad\qquad
J \overset{\text{def}}{=} (x = 0) \land (y = 0)$$

Figure II.4 A simple derivation in the invariant-based program logic

8 Proofs

In this section, we present notable proofs of the results presented in Section 7. We first present our thread-local simulation relation for proving the soundness of transformations (Section 8.1). Then we present the proofs of DRF-RA (Section 8.2) and DRF-LOCK (Section 8.3), which are the only theorems not formalized in Coq.

8.1 Thread-Local Simulation Relation

As discussed in Section 7.1, our thread-local simulation relation is based on novel ideas for reasoning about the shared resources and consistency. Now we discuss each point in more details and present the simulation's definition.

Machine Simulation Relation But before delving into the details, we first review simulation relations. Recall from Section 2.1 that a relation $\mathcal{R}$ over source and target machine states is a (machine) simulation if, for any source and target states $\mathbf{MS}_0$ and $\mathbf{MS}'_0$ related by $\mathcal{R}$, the following holds:

(**Step**) For all $\mathbf{MS}'_1$, if $\mathbf{MS}'_0 \to \mathbf{MS}'_1$, then there exists $\mathbf{MS}_1$ such that $\mathbf{MS}_0 \to^* \mathbf{MS}_1$ and $\mathcal{R}$ relates $\mathbf{MS}_1$ and $\mathbf{MS}'_1$; and

(**Terminal**) If $\mathbf{MS}'_0$ is terminal, then there exists $\mathbf{MS}_1$ such that $\mathbf{MS}_0 \to^* \mathbf{MS}_1$ and $\mathbf{MS}_1$ is terminal.

Here, we intentionally omitted events in the definition of simulation relation for presentational purposes, but the discussion in this section applies also to simulation relations with events. Also recall that if two states are simulated, then the behavior of the target state is a subset of that of the source state.

Reasoning about Shared Resources Now we are about to "localize" the definition of simulation relation to thread configurations. Our goal is proving compositionality of thread-local simulation: if $\langle \mathcal{TS}, \mathcal{S}, M \rangle$ and $\langle \mathcal{TS}', \mathcal{S}', M' \rangle$ are source and target machine states, and for each i, there exists a thread-local simulation relation R_i that relates $\langle \mathcal{TS}(i), \mathcal{S}, M \rangle$ and $\langle \mathcal{TS}'(i), \mathcal{S}', M' \rangle$, then there exists a machine simulation relation $\mathcal{R}$ that relates $\langle \mathcal{TS}, \mathcal{S}, M \rangle$ and $\langle \mathcal{TS}', \mathcal{S}', M' \rangle$.

A naive attempt to define thread-local simulation would be directly translating machine simulation relation to thread configurations. However, it does not satisfy compositionality because, in a thread's point of view, the other threads may change the shared resources at any time. For example, for machine states $\langle \mathcal{TS}_0, \mathcal{S}_0, M_0 \rangle$ and $\langle \mathcal{TS}'_0, \mathcal{S}'_0, M'_0 \rangle$, if $\mathcal{TS}_0(1)$ and $\mathcal{TS}'_0(1)$ take steps, the shared resources $\mathcal{S}_0$, M_0, $\mathcal{S}'_0$, and M'_0 may be changed, say to $\mathcal{S}_1$, M_1, $\mathcal{S}'_1$, and M'_1, and thus $\langle \mathcal{TS}_0(2), \mathcal{S}_1, M_1 \rangle$ and $\langle \mathcal{TS}'_0(2), \mathcal{S}'_1, M'_1 \rangle$ are no longer thread-locally simulated.

Thus in the definition of thread-local simulation, we need to take into account the effect—or the interference—of the other threads to the shared resources at any time. Our key observation is that the interference of the other threads can be abstractly characterized as follows. Suppose $\langle TS_0, \mathcal{S}_0, M_0 \rangle$ and $\langle TS'_0, \mathcal{S}'_0, M'_0 \rangle$ are the source and target thread configurations in our concern, and the other threads changed the source and target shared resources to $\langle \mathcal{S}_1, M_1 \rangle$ and $\langle \mathcal{S}'_1, M'_1 \rangle$. Then the following holds:

(Intrf-Future) $\langle \mathcal{S}_1, M_1 \rangle$ is a future shared resource of $\langle \mathcal{S}_0, M_0 \rangle$ w.r.t. $TS_0.\texttt{prm}$, or in other words, $\mathcal{S}_1 \sqsupseteq \mathcal{S}_0$ and M_1 is a future memory of M_0 w.r.t. $TS_0.\texttt{prm}$. Similarly, $\langle \mathcal{S}'_1, M'_1 \rangle$ is a future shared resource of $\langle \mathcal{S}'_0, M'_0 \rangle$ w.r.t. $TS'_0.\texttt{prm}$;

(Intrf-SC) If $\mathcal{S}_0 \sqsubseteq \mathcal{S}'_0$, then $\mathcal{S}_1 \sqsubseteq \mathcal{S}'_1$; and

(Intrf-Memory) If $M_0 \sim M'_0$, then $M_1 \sim M'_1$. Here, by $M \sim M'$ we mean (1) for each location, the timestamps covered by the messages in M coincides with those covered by the messages in M'; and (2) for each message $\langle x : v@(f', t], R' \rangle$ in M', there exists a message $\langle x : v@(f, t], R \rangle$ in M such that $R \sqsubseteq R'$.

From now on, we write $\langle \mathcal{S}, M \rangle \sim \langle \mathcal{S}', M' \rangle$ to mean $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $M \sim M'$ for simplicity.

Now in proving thread-local simulations, we *assume* that the interference of the other threads are constrained by the above conditions; on the other hand, we also *guarantee* that the thread in our concern also satisfies the above conditions. Concretely, we revise our thread-local simulation relation as follows. A relation R over source and target thread configurations is a thread-local simulation if:

$$\forall \langle TS_0, \mathcal{S}_0, M_0\rangle, \langle TS'_0, \mathcal{S}'_0, M'_0\rangle.\ R\ \langle TS_0, \mathcal{S}_0, M_0\rangle\ \langle TS'_0, \mathcal{S}'_0, M'_0\rangle \implies$$

$$\forall \langle \mathcal{S}_1, M_1\rangle, \langle \mathcal{S}'_1, M'_1\rangle.\ \langle \mathcal{S}_0, M_0\rangle \sim \langle \mathcal{S}'_0, M'_0\rangle \land \langle \mathcal{S}_1, M_1\rangle \sim \langle \mathcal{S}'_1, M'_1\rangle \land$$

$$\langle \mathcal{S}_1, M_1\rangle \text{ is a future of } \langle \mathcal{S}_0, M_0\rangle \text{ w.r.t. } TS_0 \land$$

$$\langle \mathcal{S}'_1, M'_1\rangle \text{ is a future of } \langle \mathcal{S}'_0, M'_0\rangle \text{ w.r.t. } TS'_0 \implies$$

(Step) For any $\langle TS'_2, \mathcal{S}'_2, M'_2\rangle$, if $\langle TS'_0, \mathcal{S}'_1, M'_1\rangle \to \langle TS'_2, \mathcal{S}'_2, M'_2\rangle$, then there exists $\langle TS_2, \mathcal{S}_2, M_2\rangle$ such that $\langle TS_0, \mathcal{S}_1, M_1\rangle \to^* \langle TS_2, \mathcal{S}_2, M_2\rangle$, R relates $\langle TS_2, \mathcal{S}_2, M_2\rangle$ and $\langle TS'_2, \mathcal{S}'_2, M'_2\rangle$, and $\langle \mathcal{S}_2, M_2\rangle \sim \langle \mathcal{S}'_2, M'_2\rangle$; and

(Terminal) If $\langle TS'_0, \mathcal{S}'_1, M'_1\rangle$ is terminal, then there exists $\langle TS_2, \mathcal{S}_2, M_2\rangle$ such that $\langle TS_0, \mathcal{S}_1, M_1\rangle \to^* \langle TS_2, \mathcal{S}_2, M_2\rangle$, $\langle TS_2, \mathcal{S}_2, M_2\rangle$ is terminal, and $\langle \mathcal{S}_2, M_2\rangle \sim \langle \mathcal{S}'_1, M'_1\rangle$; and $\cdots$

Notice that it differs from the machine simulation relation, except for types, in that (1) we should prove simulation after all legit interference of the other threads (lines 2-4), and (2) we should prove the thread's modification to the shared resources is legit (lines 7 and 10).

It is worth noting that the above conditions on the shared resources resemble memory invariants in (sequential) compiler verification literature, *e.g.*, memory extensions and injections in CompCert [57]. Both are indeed introduced for the same purpose, namely reasoning about the effect of unknown others, but they differ in what are the "others": it is other threads in our case, and it is other functions for *e.g.*, CompCert.

Reasoning about Consistency The above definition still does not satisfy compositionality because it fails to simulate consistency. Recall that for each machine step, the resulting thread configuration should be consistent. The above simulation relation does not necessarily mean the source thread configuration steps to a consistent one even if the corresponding target thread configuration steps to a consistent one.

To address this problem, we require the following conditions, in addition to the **(Step)** and **(Terminal)** conditions above, for thread-local simulation:

(Future) For any shared resource $\langle \mathcal{S}_2, M_2\rangle$ that is a future of $\langle \mathcal{S}_1, M_1\rangle$ w.r.t. $TS_0.\texttt{prm}$, there exists a shared resource $\langle \mathcal{S}'_2, M'_2\rangle$ that is a future of $\langle \mathcal{S}'_1, M'_1\rangle$ w.r.t. $TS'_0.\texttt{prm}$ such that R relates $\langle TS_0, \mathcal{S}_2, M_2\rangle$ and $\langle TS'_0, \mathcal{S}'_2, M'_2\rangle$; and

(No promises) If $TS'_0.\texttt{prm} = \varnothing$, then there exists $\langle TS_2, \mathcal{S}_2, M_2\rangle$ such that $\langle TS_0, \mathcal{S}_1, M_1\rangle \to^* \langle TS_2, \mathcal{S}_2, M_2\rangle$ and $TS_2.\texttt{prm} = \varnothing$.

Provided that the above conditions hold, consistency is also simulated as follows. Suppose that a thread-local simulation R relates source and target thread configurations $\langle TS_0, \mathcal{S}_0, M_0 \rangle$ and $\langle TS'_0, \mathcal{S}'_0, M'_0 \rangle$, and the target configuration is consistent. Now we prove $\langle TS_0, \mathcal{S}_0, M_0 \rangle$ is also consistent. For any future shared resources $\mathcal{S}_1, M_1$ of $\mathcal{S}_0, M_0$ w.r.t. $TS_0.\texttt{prm}$, by (Future), there exist $\mathcal{S}'_1, M'_1$ such that they are a future of $\mathcal{S}'_0, M'_0$ w.r.t. $TS'_0.\texttt{prm}$ and R relates $\langle TS_0, \mathcal{S}_1, M_1 \rangle$ and $\langle TS'_0, \mathcal{S}'_1, M'_1 \rangle$. Since $\langle TS'_0, \mathcal{S}'_0, M'_0 \rangle$ is consistent, there exists $\langle TS'_2, \mathcal{S}'_2, M'_2 \rangle$ such that $\langle TS'_0, \mathcal{S}'_1, M'_1 \rangle \to^* \langle TS'_2, \mathcal{S}'_2, M'_2 \rangle$ and TS'_2 has no promises. By (Step), there exists $\langle TS_2, \mathcal{S}_2, M_2 \rangle$ such that $\langle TS_0, \mathcal{S}_1, M_1 \rangle \to^* \langle TS_2, \mathcal{S}_2, M_2 \rangle$ and R relates $\langle TS_2, \mathcal{S}_2, M_2 \rangle$ and $\langle TS'_2, \mathcal{S}'_2, M'_2 \rangle$. By (No promises), there exists $\langle TS_3, \mathcal{S}_3, M_3 \rangle$ such that $\langle TS_2, \mathcal{S}_2, M_2 \rangle \to^* \langle TS_3, \mathcal{S}_3, M_3 \rangle$ and TS_3 has no promises. Thus $\langle TS_0, \mathcal{S}_0, M_0 \rangle$ is consistent.

After taking into account the effect of the other threads to the shared resources and consistency, we can prove thread-local simulation's compositionality by coinduction.

8.2 Proof of DRF-RA

We first define the set of memory events $\alpha \in \text{ME}$ as follows:

$$
\begin{aligned}
&\{\,\texttt{silent}\,\}\\
\cup\,&\{\,\texttt{read}(o, x, t) \mid o \in \text{AM}, x \in \mathsf{Loc}, t \in \mathsf{Time}\,\}\\
\cup\,&\{\,\texttt{write}(o, x, t) \mid o \in \text{AM}, x \in \mathsf{Loc}, t \in \mathsf{Time}\,\}\\
\cup\,&\{\,\texttt{update}(o_\text{r}, o_\text{W}, x, t_\text{r}, t_\text{W}) \mid o_\text{r}, o_\text{W} \in \text{AM}, x \in \mathsf{Loc}, t_\text{r}, t_\text{W} \in \mathsf{Time}\,\}\\
\cup\,&\{\,\texttt{fence}(T) \mid T \in \{\texttt{acq}, \texttt{rel}, \texttt{sc}\}\,\}\\
\cup\,&\{\,\texttt{syscall}(v) \mid v \in \ldots\,\}
\end{aligned}
$$

where $\text{AM} = \{\texttt{pln}, \texttt{rlx}, \texttt{ra}\}$.

We call the following events *globally synchronizing*:

$$
\begin{aligned}
&\{\,\texttt{fence}(\texttt{sc})\,\}\\
\cup\,&\{\,\texttt{syscall}(v) \mid v \in \ldots\,\}
\end{aligned}
$$

Then we annotate promise-free and release-acquire steps with the executed thread ids and memory events, denoted $\Longrightarrow_{(i,\alpha)}$ and $\overset{\mathsf{ra}}{\Longrightarrow}_{(i,\alpha)}$.

First, we prove two key lemmas for $\overset{\mathsf{ra}}{\Longrightarrow}$: one for removing an intermediate step, and another for reordering adjacent steps.

Lemma 5 (Step removal)**.** Suppose we have a release-acquire execution

$$\mathbf{MS} \overset{\mathsf{ra}}{\Longrightarrow}_{(i_1,\alpha_1)} \mathbf{MS}_1 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \cdots \overset{\mathsf{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}_n$$

such that

$$\forall k \geq 2.\ \mathbf{MS}_k.\mathtt{ths}(i_k).\mathtt{view} \not\geq \mathbf{MS}_1.\mathtt{ths}(i_1).\mathtt{view}\,.$$

Then we have $i_k \neq i_1$ for all $k \geq 2$ and the following execution

$$\mathbf{MS} \overset{\mathsf{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \mathbf{MS}'_2 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_3,\alpha_3)} \cdots \overset{\mathsf{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}'_n$$

for some machine states $\mathbf{MS}'_k$ satisfying

$$\forall k \geq 2.\ \forall i \neq i_1.\ \mathbf{MS}'_k.\mathtt{ths}(i).\mathtt{st} = \mathbf{MS}_k.\mathtt{ths}(i).\mathtt{st}\,.$$

Proof. There are only two cases where the first step with (i_1, α_1) affects a subsequent step with (i_k, α_k): either (i) the latter reads what the former wrote; or (ii) the former globally synchronizes. In case (i), the view $\mathbf{MS}_k.\mathtt{ths}(i_k).\mathtt{view}$ becomes as high as the view $\mathbf{MS}_1.\mathtt{ths}(i_1).\mathtt{view}$ because the read and write are ra-synchronized. This is impossible because it conflicts with the assumption. In case (ii), the effect is limited: $\mathbf{MS}'_k$ is the same as $\mathbf{MS}_k$ except the effect of the event α_1. More specifically, $\mathbf{MS}_k$'s memory may contain an extra message produced by α_1 and the threads other than i_1 in $\mathbf{MS}'_k$ are the same as those in $\mathbf{MS}_k$ except that every view in the former may be less than the corresponding view in the latter. By monotonicity, $\mathbf{MS}'_k$ has more behaviors than $\mathbf{MS}_k$ and thus we can construct such an execution. □

Lemma 6 (Step reorder)**.** Suppose we have a release-acquire execution

$$\mathbf{MS} \overset{\mathsf{ra}}{\Longrightarrow}_{(i_1,\alpha_1)} \mathbf{MS}_1 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \mathbf{MS}_2$$

such that one of α_1 and α_2 is not globally synchronizing and

$$\mathbf{MS}_1.\mathtt{ths}(i_1).\mathtt{view} \not\leq \mathbf{MS}_2.\mathtt{ths}(i_2).\mathtt{view}\,.$$

Then we have $i_1 \neq i_2$ and $\mathbf{MS}'_1$ satisfying

$$\mathbf{MS} \overset{\mathsf{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \mathbf{MS}'_1 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_1,\alpha_1)} \mathbf{MS}_2\,.$$

Proof. Basically a similar argument as in the previous lemma applies here: (i) α_2 should not read α_1; and (ii) the earlier step with α_2 does not affect the later step with α_1 since α_1 or α_2 is not globally synchronizing. □

Now we prove DRF-RA. Let $\overset{\mathsf{ra}}{\Longrightarrow}$ be identical to $\Longrightarrow$ in Theorem 1, except that (i) `rlx` and `pln` accesses in program transitions are interpreted as if they are all `ra`-accesses, and (ii) a machine step consists only of one thread step. Note that the second condition does not affect the semantics, since a machine state without promises is vacuously consistent.

Proof of DRF-RA (Theorem 2). It suffices to show that (i) the existence of a $\mathtt{rlx}$-race in the $\Longrightarrow$-machine implies that in the $\overset{\mathsf{ra}}{\Longrightarrow}$-machine, and (ii) the behavior in the $\overset{\mathsf{ra}}{\Longrightarrow}$-machine and that in the $\Longrightarrow$-machine coincide if $\mathcal{P}$ is $\mathtt{rlx}$-race-free in the $\overset{\mathsf{ra}}{\Longrightarrow}$-machine. Then Theorem 1 concludes the proof.

We prove both (i) and (ii) by a single simulation argument. We say an $\overset{\mathsf{ra}}{\Longrightarrow}$-execution

$$\mathbf{MS}'_0 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_1,\alpha_1)} \mathbf{MS}'_1 \overset{\mathsf{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \cdots \overset{\mathsf{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}'_n$$

simulates a $\Longrightarrow$-execution

$$\mathbf{MS}_0 \Longrightarrow_{(i_1,\alpha_1)} \mathbf{MS}_1 \Longrightarrow_{(i_2,\alpha_2)} \cdots \Longrightarrow_{(i_n,\alpha_n)} \mathbf{MS}_n \ ,$$

if the following conditions hold:

1. $\forall k, j.\ \mathbf{MS}_k.\mathtt{ths}(j).\mathtt{st} = \mathbf{MS}'_k.\mathtt{ths}(j).\mathtt{st}$;
2. $\forall k, j.\ \mathbf{MS}_k.\mathtt{ths}(j).\mathtt{view.cur} = \mathbf{MS}'_k.\mathtt{ths}(j).\mathtt{view.cur}$;
3. $\forall k, j.\ \mathbf{MS}_k.\mathtt{ths}(j).\mathtt{view.acq} = \mathbf{MS}'_k.\mathtt{ths}(j).\mathtt{view.acq}$;
4. $\forall k.\ \mathbf{MS}_k.\mathtt{gsc} = \mathbf{MS}'_k.\mathtt{gsc}$; and
5. $\forall k.\ \mathbf{MS}_k.\mathtt{mem}$ and $\mathbf{MS}'_k.\mathtt{mem}$ have the same messages, except that the released view of a message in the $\overset{\mathsf{ra}}{\Longrightarrow}$-execution may be higher than that of the corresponding message in the $\Longrightarrow$-execution.

This simulation proves (i), as a $\mathtt{rlx}$-race in $\mathbf{MS}_k$ in the $\Longrightarrow$-execution is also a $\mathtt{rlx}$-race in $\mathbf{MS}'_k$ in the $\overset{\mathsf{ra}}{\Longrightarrow}$-machine, thanks to the condition 1. It also proves (ii) by the adequacy of the simulation relation.

Now we prove the simulation. Consider a $\Longrightarrow$-step:

$$\mathbf{MS}_n \Longrightarrow_{(i_{n+1},\alpha_{n+1})} \mathbf{MS}_{n+1} \ ,$$

and we will find a corresponding $\overset{\mathsf{ra}}{\Longrightarrow}$-step that preserves the simulation relation:

$$\mathbf{MS}'_n \overset{\mathsf{ra}}{\Longrightarrow}_{(i_{n+1},\alpha_{n+1})} \mathbf{MS}'_{n+1} \ .$$

Thanks to the simulation relation, there exists $\mathbf{MS}'_{n+1}$ such that $\mathbf{MS}'_n \overset{\mathsf{ra}}{\Longrightarrow}_{(i_{n+1},\alpha_{n+1})} \mathbf{MS}'_{n+1}$. If α_{n+1} is not reading (*i.e.*, neither a read nor an update event), it is immediate from the semantics that the simulation relation is preserved. Now suppose α_{n+1} is reading $\langle x@t \rangle$ with the access mode o_r, and let k be such an index that α_k is writing $\langle x@t \rangle$ with the access mode o_w, and R (and R_{ra}) be the released view of $\langle x@t \rangle$ in the $\Longrightarrow$-execution (and $\overset{\mathsf{ra}}{\Longrightarrow}$-execution, respectively).

Now we proceed by a case analysis:

Case $o_w, o_r \sqsupseteq$ `ra`.

Note that the current & acquire views of $\mathbf{MS}_{n+1}.\texttt{ths}(i_{n+1})$ and $\mathbf{MS}'_{n+1}.\texttt{ths}(i_{n+1})$ may diverge only due to the discrepancy of the $\langle x@t\rangle$'s released views (R and $R_{\texttt{ra}}$), and the read's access mode (o_r for the $\Rightarrow$-machine, and $o_r \sqcup \texttt{ra}$ for the $\overset{\texttt{ra}}{\Longrightarrow}$-machine). A similar argument applies to the other machine state components in the simulation relation. Hence it suffices to show that $R = R_{\texttt{ra}}$ and $o_r = o_r \sqcup \texttt{ra}$, which come clearly from the assumption: in particular we have $R = \mathbf{MS}_k.\texttt{ths}(i_k).\texttt{view}.\texttt{cur} = \mathbf{MS}'_k.\texttt{ths}(i_k).\texttt{view}.\texttt{cur} = R_{\texttt{ra}}$ thanks to $o_w \sqsupseteq$ `ra`.

Case $\mathbf{MS}'_k.\texttt{ths}(i_k).\texttt{view}.\texttt{cur} \leq \mathbf{MS}'_n.\texttt{ths}(i_n).\texttt{view}.\texttt{cur}$.

Since the released view $R_{\texttt{ra}} = \mathbf{MS}'_k.\texttt{ths}(i_k).\texttt{view}.\texttt{cur}$ of $\langle x@t\rangle$ is already incorporated in the current view, a similar argument also applies here.

Otherwise.

In this case, we construct a `rlx`-race in the $\overset{\texttt{ra}}{\Longrightarrow}$-execution by repeatedly applying the step-removing Lemma 5 to the execution:

$$\mathbf{MS}'_{k-1} \overset{\texttt{ra}}{\Longrightarrow}_{(i_k,\alpha_k)} \mathbf{MS}'_k \overset{\texttt{ra}}{\Longrightarrow}_{(i_{k+1},\alpha_{k+1})} \cdots \overset{\texttt{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}'_n \,,$$

so that i_k (attempting to write to x with o_w) and i_{n+1} (attempting to read from x with o_r) race in a reachable machine state.

Let $j \in [k, n)$ be the last such an index that $\mathbf{MS}'_j.\texttt{ths}(i_j).\texttt{view}.\texttt{cur} \nleq \mathbf{MS}'_n.\texttt{ths}(i_n).\texttt{view}.\texttt{cur}$. By Lemma 5 there exists an $\overset{\texttt{ra}}{\Longrightarrow}$-execution:

$$\mathbf{MS}'_{j-1} \overset{\texttt{ra}}{\Longrightarrow}_{(i_{j+1},\alpha_{j+1})} \mathbf{MS}''_{j+1} \cdots \overset{\texttt{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}''_n \,,$$

such that $\mathbf{MS}''_n.\texttt{ths}(i_{n+1}).\texttt{st} = \mathbf{MS}'_n.\texttt{ths}(i_{n+1}).\texttt{st}$. By repetition, we have an $\overset{\texttt{ra}}{\Longrightarrow}$-execution:

$$\mathbf{MS}'_{k-1} \overset{\texttt{ra}}{\Longrightarrow}_{(i_l,\alpha_l)} \mathbf{MS}'''_l \overset{\texttt{ra}}{\Longrightarrow}_{(i_u,\alpha_u)} \cdots \overset{\texttt{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}'''_n \,,$$

such that $\mathbf{MS}'''_n.\texttt{ths}(i_{n+1}).\texttt{st} = \mathbf{MS}'_n.\texttt{ths}(i_{n+1}).\texttt{st}$ and i_k is not executed at all from $\mathbf{MS}'_{k-1}$ to $\mathbf{MS}'''_n$. Hence $\mathbf{MS}'''_n.\texttt{ths}(i_k).\texttt{st} = \mathbf{MS}'_{k-1}.\texttt{ths}(i_k).\texttt{st}$, thus i_k and i_{n+1} race in $\mathbf{MS}'''_n$.

□

8.3 Proof of DRF-LOCK

To state a DRF theorem on properly locked programs, we classify locations into normal locations and *lock locations* and suppose lock locations are accessed only by the acquire and release operations, as defined as follows:

```
acquire(l) {                                      release(l) {
   while !CAS(l, 0, 1, acqrel) do skip;              l_rel := 0;
}                                                 }
```

Furthermore, we say a machine state **MS** is *properly locked*, if:

1. If two different threads can take a step accessing the same location, then both accesses are reads, or the location is a lock location; and

2. If a thread can release a lock, say l, then the value of l in **MS**'s memory is 1.

Theorem 7 (DRF-LOCK)**.** Let $\overset{\texttt{sc}}{\Longrightarrow}$ denote the steps of the interleaving machine. Suppose that every machine state that is $\overset{\texttt{sc}}{\Longrightarrow}$-reachable from the initial state of a program $\mathcal{P}$ is properly locked. Then, the behaviors of $\mathcal{P}$ according to the full machine coincide with those according to the $\overset{\texttt{sc}}{\Longrightarrow}$-machine.

Proof. We say that an $\overset{\texttt{ra}}{\Longrightarrow}$-execution is *interleaving* if any reading step reads from the message with the greatest timestamp and any writing step writes a message with a timestamp greater than any existing message's timestamp. It is obvious that the $\overset{\texttt{sc}}{\Longrightarrow}$-machine is equivalent to the interleaving $\overset{\texttt{ra}}{\Longrightarrow}$-machine (which we simply call the interleaving machine), and thus we identify them.

First of all, for any $\overset{\texttt{ra}}{\Longrightarrow}$-execution E (*i.e.*, a finite or infinite sequence of $\overset{\texttt{ra}}{\Longrightarrow}$-steps), it is easy to see that removing all failed acquire steps from the execution still yields a valid $\overset{\texttt{ra}}{\Longrightarrow}$-execution E' with the same behavior (*i.e.*, the same sequence of system calls). Furthermore, if E is a finite execution leading to a `ra`-racy machine state, then so is E'. We will simply say $\overset{\texttt{nfra}}{\Longrightarrow}$-executions for $\overset{\texttt{ra}}{\Longrightarrow}$-executions with no failed acquire steps.

From this observation and Theorem 2, we can easily see that it suffices to prove that (i) the existence of a `ra`-race in an $\overset{\texttt{nfra}}{\Longrightarrow}$-execution of $\mathcal{P}$ implies a violation of proper locking in an interleaving execution of $\mathcal{P}$; and (ii) the $\overset{\texttt{nfra}}{\Longrightarrow}$-behaviors of $\mathcal{P}$ coincide with its interleaving behaviors if $\mathcal{P}$ is properly locked in all interleaving executions.

We prove both (i) and (ii) by a single simulation argument. We say an interleaving execution

$$\mathbf{MS}'_0 \overset{\texttt{ra}}{\Longrightarrow}_{(i'_1,\alpha'_1)} \mathbf{MS}'_1 \overset{\texttt{ra}}{\Longrightarrow}_{(i'_2,\alpha'_2)} \cdots \overset{\texttt{ra}}{\Longrightarrow}_{(i'_n,\alpha'_n)} \mathbf{MS}'_n$$

simulates an $\overset{\texttt{nfra}}{\Longrightarrow}$-execution

$$\mathbf{MS}_0 \overset{\texttt{ra}}{\Longrightarrow}_{(i_1,\alpha_1)} \mathbf{MS}_1 \overset{\texttt{ra}}{\Longrightarrow}_{(i_2,\alpha_2)} \cdots \overset{\texttt{ra}}{\Longrightarrow}_{(i_n,\alpha_n)} \mathbf{MS}_n \ ,$$

if the following conditions hold:

1. $(i'_1, \alpha'_1), \ldots, (i'_n, \alpha'_n)$ is a reordering of $(i_1, \alpha_1), \ldots, (i_n, \alpha_n)$ such that the order of system calls is preserved; and
2. $\mathbf{MS}_0 = \mathbf{MS}'_0$ and $\mathbf{MS}'_n = \mathbf{MS}_n$.

If we prove that given any $\overset{\texttt{nfra}}{\Longrightarrow}$-execution of length n there exists a simulating interleaving execution, then we are done as follows. Given any (possibly infinite) $\overset{\texttt{nfra}}{\Longrightarrow}$-execution and any number of steps n, we can find an interleaving execution of length n leading to the same machine state with the same sequence of observable events (*i.e.*, system calls). Thus, any arbitrarily long observation on an $\overset{\texttt{nfra}}{\Longrightarrow}$-execution cannot be distinguished from that on an interleaving execution. Also, if there is any $\overset{\texttt{nfra}}{\Longrightarrow}$-execution leading to a `ra`-racy machine state, we can find a simulating interleaving execution to an improperly locked machine state by the simulation argument.

Now it suffices to prove the simulation theorem by induction on the length n. The base case is trivial. For an induction step, let's assume that we have a simulating execution of length n, given as in the above definition of simulation. Suppose we have a step $\mathbf{MS}_n \overset{\texttt{nfra}}{\Longrightarrow}_{i_{n+1},\alpha_{n+1}} \mathbf{MS}_{n+1}$. Then we need to find a simulating interleaving execution of length $n+1$ that starts from $\mathbf{MS}_0$ and ending in $\mathbf{MS}_{n+1}$. If the event α_{n+1} is neither a read, a write, nor an update, then the execution $\mathbf{MS}'_0 \ldots \mathbf{MS}'_n \overset{\texttt{ra}}{\Longrightarrow}_{(i_{n+1},)} \mathbf{MS}_{n+1}$ is interleaving, so we are done.

Thus suppose that α_{n+1} is accessing (*i.e.*, a read, a write, or an update event on) a location x and does not satisfy the interleaving condition (*i.e.*, does not read the latest message nor writes with a greatest timestamp). By definition of the interleaving condition, we can find an event writing to x whose timestamp is bigger than that of the event α_{n+1}. Let's write α_k and t_k for the first such event (*i.e.*, with the smallest index k) and its timestamp.

Now, by exactly the same argument as in Theorem 2, we can remove all steps $\mathbf{MS}_j$ such that $k \le j \le n$ and $\mathbf{MS}_j.\texttt{ths}(i_j).\texttt{view} \ge \mathbf{MS}_k.\texttt{ths}(i_k).\texttt{view}$. Then we have a race between α_k and α_{n+1}. The resulting execution is also interleaving because removing a step from an interleaving execution always results in an interleaving execution. Thus, by the proper locking assumption, it follows that α_k and α_{n+1} are accessing the same lock location.

Now we will construct an interleaving execution from $\mathbf{MS}_0$ to $\mathbf{MS}_{n+1}$ that simulates the given execution. For this, we repeatedly apply the step-reordering using Lemma 6 as follows. First, we find the first event, say α_j, such that $k \leq j \leq n{+}1$ and $\mathbf{MS}_j.\texttt{ths}(i_j).\texttt{view.cur.rlx}(x) < t_k$. Then we can move down the event to just before α_k using Lemma 6. We repeat this process until we move α_{n+1} down to just before α_k. This is possible because we have $\mathbf{MS}_{n+1}.\texttt{ths}(i_{n+1}).\texttt{view.cur.rlx}(x) < t_k$. Also note that this process does not reorder system calls because we assume that system calls synchronize on lock locations (*i.e.*, making the view on lock locations to be up-to date).

Finally we will show that such a reordering does not break the interleaving condition for all existing events and furthermore make α_{n+1} to satisfy the interleaving condition. The latter holds trivially by construction because α_k was the first event with respect to which α_{n+1} violates the interleaving condition. The former holds as follows. In order to break the interleaving condition, we need to reorder two events α, β to β, α such that they are accessing the same location and at least one of them is writing. During the reordering process, suppose we meet such a reordering for the first time. Then, the execution before the reordering is interleaving because we are about to break the condition for the first time. Since α and β are racing on the same location, they both have to be lock operations (*i.e.*, successful acquire or release). By the proper locking assumption, we have only two possibilities: α is a release and β is an acquire; or α is an acquire and β is a release. The former case is a contradiction because β reads what α writes due to the interleaving condition, which makes β's view as high as α's. The latter is a contradiction too because after reordering the execution up to β is still interleaving but the machine state before β is not properly locked. Thus we can conclude that the reorderings do not break the interleaving condition.

□

9 Related Work

There have been many proposals for solving the "out of thin air" problem. Several of them have come with proofs of DRF guarantees, but ours is the first to come with formal (and machine-checked) validation of a wide range of essential local transformations (Section 7.1) concerning a broad spectrum of features from the C/C++ model.

The first major attempt to solve the "out of thin air" problem was by the Java memory model (JMM) [59] (see also [58]). The JMM intended to validate all the compiler optimizations that Java compilers and just-in-time compilers might perform, but its formal definition failed to validate them [72]. Subsequent fixes were proposed to the model, which improved the set of enabled optimizations, but still falling short of what actual Java compilers were performing.

Interestingly, an early glimpse of our idea of promises may be seen in version 1.0 of the JMM [28], which describes a form of "prescient store actions" (§17.8). However, their description is very brief and vague, and the feature was removed for JSR 133 [9].

To resolve some of the problems with the JMM definition, Jagadeesan *et. al.* [38] proposed an operational model following quite closely the intended behavior of the JMM, but employing the notion of a *speculation*. Speculations are similar to our notion of promises, but unlike promises they are not certified thread-locally: whereas we model interference conservatively by quantifying over all future memories during certification, they model interference from other threads more precisely by executing multiple threads together during certification. We believe our conservative approach is sufficient for justifying standard compiler optimizations, which are typically thread-local, and moreover it simplifies the presentation of the semantics and the development of the metatheory because it avoids the need for nested certifications.

Jagadeesan *et. al.* 's model satisfies the standard DRF theorem, as well as a DRF theorem saying that speculations are unnecessary for programs without read-write races. They also develop a simulation proof technique, with which they verify three optimizations: write-write reordering, roach-motel reordering, and read-after-read elimination. We have applied our simulation method to a much wider variety of optimizations, and our proofs are machine-checked in Coq. They also do not provide any compilation correctness results, and their model omits release-acquire accesses, updates, and fences.

More recently, Jeffrey and Riely [39] presented a weak memory model based on event structures. Their model admits a standard DRF theorem, but does not fully allow the reordering of independent memory accesses, and thus cannot be compiled to Power/ARM without extra fences. The paper suggests an idea about how to fix the model to support such reorderings, but it is not known whether the suggested fixed model avoids OOTA behaviors. Relating to our work, their model seems to be "promising" reads (instead of writes) and restricting the quantification over possible futures to only those that could arise from further execution of the current program. The model only supports relaxed accesses and locks.

Pichon-Pharabod and Sewell [67] introduced an event structure model with both a normal reduction rule, which executes an initial event of the event structure, and special reduction rules that mimic the effect of standard compiler optimizations on the event structure. These optimization rules include a rather complex rule for non-thread-local optimizations that can declare a whole branch of the event structure unreachable. The paper does not present any formal results about the model. It is worth noting that the model does not support the weak behavior of the ARM-weak program and thus may not be compiled to ARM without additional fences. The model only handles relaxed and non-atomic accesses and locks.

Podkopaev *et. al.* [69] proposed an operational model covering a large subset of the features of the C/C++ model. They provide many litmus tests to demonstrate the suitability of their model, but do not prove any formal results about it. Their model ensures per-location coherence in a very similar way to our model: using timestamps. In order to handle read-write reorderings, they allow reads to return symbolic values, which are then evaluated at a later point in time when their value is actually needed. This approach gives the expected behaviors to the LB and LBd programs, and may be extended with a set of *syntactic* symbolic simplification rules to also give the expected result to the LBfd program. It seems, however, very difficult to extend this approach to enable code motion optimizations, where some common code is pulled out of two branches of a conditional. What makes code motion more challenging is that the common code may become apparent only after some earlier transformations, like for example the $y := 1$ assignment in the following code:

$$\left.\begin{array}{l} a := x;\ \ /\!/\ 1 \\ \textbf{if } a = 1 \textbf{ then } y := a;\ \textbf{else } (z := 1; y := z;) \end{array}\right\|\; x := y;$$

Our model allows the annotated behavior of the program above, precisely because our promises are semantic in nature and thus avoid the brittle tracking of syntactic data dependencies.

Zhang and Feng [82] suggested an operational model for Java accesses in which threads may re-execute some memory events. The model admits a standard DRF theorem, and its replay mechanism enables it to support local transformations. However, to avoid OOTA, this mechanism is limited by its tracking of syntactic dependencies between instructions, and thus it fails to validate behaviors resulting from trace-preserving transformations like the one above.

Other proposals for language-level memory models have tried not to solve the OOTA problem, but to avoid it, by introducing stronger models where read-write reordering is not allowed. For example, Ševčík *et. al.* [73] and Demange *et. al.* [24] proposed using TSO as the memory model for C and Java, respectively. These proposals may be reasonable compromises if the only target machines of interest also follow the TSO model, but are prohibitively expensive on weaker architectures, such as Power and ARM, because enforcing TSO on those machines requires essentially as many fences as enforcing SC. In a similar line of work, Lahav *et. al.* [51] introduced a strengthening of the release/acquire fragment of the C/C++ memory model, which they called SRA, together with an operational semantics for SRA. Compiling SRA to Power and ARM is cheaper than TSO, but still requires some fences before or after every shared variable access, and may thus not be suitable for performance-critical code. Dolan *et. al.* [25] and Ou *et. al.*

[64] proposed C/C++, OCaml, and Java memory models that specifically forbid read-write reordering, and proposed their compilation schemes to TSO, ARM, and Power that insert less more fences than SRA. They reported the performance overhead of fences is none for TSO and within 3% on average for ARM and Power architectures. However, more comprehensive performance study is necessary to draw a solid conclusion.

Boehm advocated stronger models without read-write reorderings because they break modularity and useful reasoning principles for programmers [19]. Notwithstanding his criticism, we think it is unclear whether forbidding read-write reorderings altogether is adequately striking the trade-off between simplicity and performance. To find a sweet spot, we are developing reasoning principles and tools for programmers in the presence of read-write reorderings. See Section 10 for more details.

Another approach is to simply allow OOTA behaviors. This was the approach taken by Batty *et. al.* 's formalization of C/C++ [13], and by the OpenCL model [44], as well as by Crary and Sullivan [23], who introduced a more fine-grained specification of the orders that the model is supposed to preserve. All of these models allow the weak behavior of the LBd example, thereby invalidating standard reasoning principles and DRF theorems.

Finally, Norris and Demsky [62] presented a tool that exhaustively enumerates the behaviors of concurrent C/C++ programs. To account for speculative reads, the tool may establish "promised future values", which a load can read from, and which must eventually be written by a future store. Norris and Demsky's promises look superficially quite similar to ours, but their purpose is to support practical model checking of C/C++ programs, not to change the semantics of the language, so the paper does not present any formal model or metatheory of promises.

10 Follow-up and Future Work

SC Accesses Lahav *et. al.* proposed a fix to SC accesses in C/C++11 [53]. Extending our model with SC accesses is left for future work (see Section 3.3).

Compilation Correctness Podkopaev *et. al.* established the correctness of compilation of our model to Power, ARMv7, and RISC-V, as well as to ARMv8 using a sub-optimal compilation scheme for read-modify-update instructions [68]. In fact, the optimal one used by mainstream compilers is unsound for our semantics due to the fact that ARMv8's read-modify-update instructions perform "global" optimizations, which is beyond the reach of our semantics as we will explain shortly.

Global Optimizations In our model, we insist that promises can always be certified thread-locally. This decision enables thread-local reasoning about our semantics and suffices to justify all the known thread-local program transformations that a compiler or the hardware may perform. It does, however, render unsound some transformations of a global nature, such as sequentialization (aka "thread inlining"), which merges threads together. To see this, consider the following:

$$\left.\begin{array}{l} a := x;\ \ /\!/\ 1 \\ \textbf{if}\ a = 0\ \textbf{then} \\ \quad x := 1; \end{array}\ \right\| \ y := x;\ \left\|\ x := y; \right. \quad \leadsto \quad \left.\begin{array}{l} a := x;\ \ /\!/\ 1 \\ \textbf{if}\ a = 0\ \textbf{then} \\ \quad x := 1; \\ y := x; \end{array}\ \right\|\ x := y;$$

This source program disallows the specified behavior because if T_1 reads 1 for x after promising $x := 1$, then it will not be able to fulfill its promise. Nevertheless, the result $a = 1$ *is* allowed in the target program (obtained by sequentializing T_1 before T_2). Here, T_1 can safely promise $y := 1$, and later read $x = 1$ from T_2's write.[4]

Furthermore, thrad-local certification also renders unsound the optimal compilation scheme to ARMv8. Consider the following example [68, Example 3.10]:

$$\left.\begin{array}{l} a := x;\ \ /\!/\ 1 \\ z := a \end{array}\ \right\|\ \begin{array}{l} b := z;\ \ /\!/\ 1 \\ c := FAA(x, 1, \textbf{rel});\ \ /\!/\ 0 \\ y := c + 1 \end{array}$$

This behavior is disallowed in our semantics, because a promise of $y = 1$—which is required for the behavior—cannot be certified in the presence of release fetch-and-add. However, this behavior is allowed in ARMv8, essentially because it allows the global optimization which (1) analyzes that `x` is modified only by the fetch-and-add instruction and `c` be the initial value 0, and (2) transforms `y:=c+1` with `y:=1`. After the optimization, $y = 1$ can be promised and the behavior is allowed.

Generalizing our semantics to support global optimizations and verifying the optimal compilation scheme to ARMv8 is left for future work.

Liveness It is natural to extend our operational model with liveness guarantees, and it is useful and interesting to study their interaction with program transformations and DRF theorems. Liveness properties are currently mostly ignored in weak memory research.

[4] Though sequentialization is a very intuitive property that one might expect a memory model to validate, we observe that TSO [65], Power [11], ARMv8 [27], Java [59], and C18/C++17 [13] (without the corrections proposed in [78]) all do not allow sequentialization.

Reasoning Principle The program logic presented in Section 7.4 only establishes the very basic sanity of our memory model. Svendsen *et. al.* [77] propopsed a separation logic for our semantics that is capable of proving the absence of out-of-thin-air behaviors for various litmus tests. Developing reasoning principles for our semantics that is capable of verifying concurrent data structures and algorithms, is a direction for future work.

Promising Semantics for Hardware Concurrency Using the main idea of our semantics, namely views and promises, Pulte *et. al.* [70] proposed the promising semantics for ARMv8/RISC-V which is a simpler and faster operational concurrency semantics than the existing models for those architectures. Developing the promising semantics for other architectures is left for future work.

Simulation and Model Checking The high degree of nondeterminism in relaxed-memory concurrency makes it hard to exhaustively explore all possible behaviors of a given program. In an ongoing work, we are developing efficient methods and tools for this purpose by leveraging the idea of views and promises to tame the nondeterminism. In particular, Pulte *et. al.* [70] developed an efficient model checker for ARMv8/RISC-V, and using this model checker, they verified realistic concurrent data structures such as Michael-Scott queue and Chase-Lev deque.

Chapter III

Separate Compilation and Linking

11 Introduction

There was a key dimension in which the verification of CompCert, as well as that of the other existing verified compilers, is not realistic—namely that, for simplicity, it only establishes the correctness of *whole-program* compilation: if CompCert is used to compile a self-contained C program consisting of a single file, then the output of CompCert preserves the semantics of that program. But clearly this does not correspond to what many clients of a verified compiler would expect. For example, it is often essential in practice to be able to compile a client module separately from the many standard libraries it depends on, yet be assured that linking the resulting binaries together will result in an executable that preserves the semantics of the linked source modules. Furthermore, it is commonplace for different modules in a program to be compiled with different sets of optimization passes turned on. Although the CompCert compiler does indeed support such forms of separate compilation, its verification statement says nothing about them.

The technical reason for this limitation is that it makes it possible for the CompCert verification to be carried out straightforwardly using closed simulations: simulations between closed (*i.e.*, self-contained, executable) programs. For each pass of the compiler, the output of the pass is shown to simulate the input of the pass, assuming and preserving whatever invariant the verifier wishes to impose on the relation between the states of the input and output. Working with closed simulations simplifies life in two ways: (1) the simulation proof for each pass can rely on whatever state invariant it chooses, independent of what invariants are used in other passes, and (2) these independent simulations collectively imply the end-to-end correctness of the whole compiler (this is sometimes called "vertical compositionality"). However, closed simulations are by definition simulations over whole program states, thus seemingly confining their applicability to the verification of whole-program compilation.

There has consequently been a great deal of work in the past several years attempting to prove compiler correctness *without* the whole-program restriction. Indeed, it turns out that even specifying, let alone verifying, when separate compilation is "correct"—often referred to as *compositional compiler correctness* [15]—is non-trivial, and has sparked a variety of interesting proposals involving technically sophisticated techniques, such as Kripke logical relations [31], multi-language semantics [66], and parametric simulations [32, 61]. All these approaches aim to achieve a highly flexible form of compositionality, guaranteeing for instance that the results of multiple different verified compilers can be correctly linked together, and that it is safe to link those results with hand-written assembly code.

It seems, however, that achieving such flexibility comes at the expense of significant complication to the proof method. For example, Perconti and Ahmed's approach [66] involves constructing logical relations over a multi-language semantics encompassing all languages used in a compiler, a considerable departure from CompCert-style verification. Neis *et. al.* 's method [61] employs a novel notion of "parametric inter-language simulations (PILS)", whose proof of the aforementioned "vertical compositionality" is highly involved [34], whereas for closed simulations it is trivial. In the context of CompCert, Stewart *et. al.* recently developed Compositional CompCert [76], a re-engineering of CompCert to support verified separate compilation, along with the ability to link C modules with assembly modules. Their approach, however, relies on a novel notion of "structured simulation" on top of a multi-language "interaction semantics" [16], which is different enough from the closed simulations employed in the CompCert verification that it required significant changes and extensions to the original proofs.

In this paper, we ask the question: If we aim somewhat lower, can we do a lot better? That is, if we pursue a more restricted notion of compositional correctness than prior work has done, can we develop a much simpler, more lightweight proof method that enables us to *reuse* existing verifications of whole-program compilation as much as possible instead of rewriting them?

Indeed, we can. In particular, we restrict attention here to verifying separate compilation for a *single* compiler. Our goal is to establish that, when different modules in a program are compiled separately by the *same* verified compiler, the linking of the resulting assembly modules preserves the semantics of the linking of the original source modules. Within the scope of this more modest but still important goal, we develop simple and effective techniques for verifying two levels of compositional correctness:

- **Compositional Correctness Level A:** Correctness of compilation is preserved when linking modules that were compiled with *the same exact compiler*.

- **Compositional Correctness Level B:** Correctness of compilation is preserved when linking modules that were compiled with the same compiler, *but possibly with different optimizations—i.e.*, different modules may be compiled with different optimization passes turned on.

Level B correctness is stronger than Level A, and correspondingly requires somewhat (but not much) more work to prove.

The key idea spanning both levels is to formulate a compositional correctness statement about a single *module M* in terms of a "contextual" correctness statement about how *M* behaves when linked with arbitrary other modules to form a complete program. The latter has the advantage of being a statement about closed (whole) programs, and as such, we can prove it using the kind of simple, closed-simulation-style proofs employed in traditional verifications of whole-program compilation, as far as the compiler's transformations and optimizations satisfy what we call *monotonicity*. This in turn enables us to significantly reuse existing compiler proofs. We believe the lightweight nature of our techniques—and the consequent ease of adapting existing verifications to use them—will make them a highly attractive option for compiler verifiers.

We demonstrate the effectiveness of our techniques by applying them to CompCert 2.4. In less than two person-months total, we adapted the existing CompCert verification to support both Level A and Level B compositional correctness, and much of that time was spent trying to understand the original CompCert proof. **The result of this effort is the first verification of separate compilation for the full CompCert compiler.** Our verification (available online [1]) is mostly the same as the original CompCert verification, and is only 2% (for Level A) or 3% (for Level B) larger than the original verification in terms of lines of Coq. Furthermore, in the course of doing so, we uncovered two bugs in CompCert: one in an invalid axiom, and one (in CompCert's "value analysis") that was outside the scope of CompCert's original verification because it only showed up in the presence of separate compilation. These have been confirmed and subsequently fixed in CompCert 2.5.

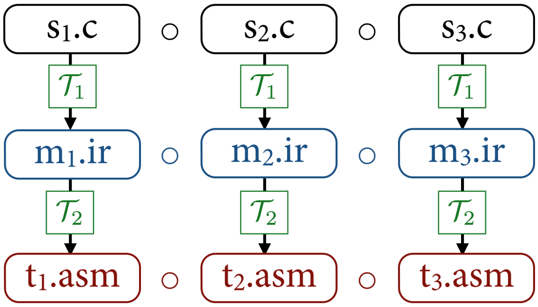


Figure III.1 Proving Level A correctness

The remainder of this chapter is structured as follows. In Section 12, we give the overview of our new techniques, which are presented in detail in the subsequent sections in the context of our CompCert adaptation. In Section 16, we conclude with a comparison to related work and discussion on the generality and the impact of our techniques.

12 Overview

We begin by explaining our Level A and Level B notions of compositional correctness and our techniques for establishing them (Section 12.1 and Section 12.2). We also briefly give some intuition as to why it is easy to adapt CompCert's verification to employ our new techniques, but we leave a more thorough explanation of this adaptation to subsequent sections. Throughout the section, we keep the presentation semi-formal, abstracting away unnecessary detail to get across the main ideas.

12.1 Compositional Correctness Level A

End-to-End Correctness For Level A, we aim to show that if we separately compile n different C modules ($\mathtt{s}_1\mathtt{.c}, \ldots, \mathtt{s}_n\mathtt{.c}$) using the same exact verified compiler $\mathcal{C}$, producing n assembly modules ($\mathtt{t}_1\mathtt{.asm}, \ldots, \mathtt{t}_n\mathtt{.asm}$), then the assembly-level linking of the $\mathtt{t}_i$'s will refine the C-level linking of the $\mathtt{s}_i$'s. Formally:

$$\frac{\forall i \in \{1 \ldots n\}.\ \mathcal{C}(\mathtt{s}_i\mathtt{.c}) = \mathtt{t}_i\mathtt{.asm} \qquad s = \mathrm{load}(\mathtt{s}_1\mathtt{.c} \circ \ldots \circ \mathtt{s}_n\mathtt{.c}) \qquad t = \mathrm{load}(\mathtt{t}_1\mathtt{.asm} \circ \ldots \circ \mathtt{t}_n\mathtt{.asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

Here, $\circ$ represents simple *syntactic* linking, *i.e.*, essentially concatenation of files (plus checks to make sure that externally declared variables/functions have the expected types). See Section 13 for further details about syntactic linking.

It is worth noting that the end-to-end correctness of whole-program copmilation defined in Section 2.1 is just a degenerate case of the Level A correctness in which there is only one source code, *i.e.*, $i = 1$.

Per-Pass Correctness To prove that compiler $\mathcal{C}$ satisfies Level A compositional correctness, we want to reduce the problem to one of verifying the individual passes of $\mathcal{C}$, as we did for the end-to-end correctness of whole-program compilation. The key idea here, as illustrated in Figure III.1 where three separately-compiled modules go through two compiler passes $\mathcal{T}_1$ and $\mathcal{T}_2$, is that since we know that the source modules are all compiled via the exact same sequence of passes, we can verify their compilations in lock step as if each pass is applied to all modules simultaneously. In other words, it suffices to verify that, for each pass $\mathcal{T}$ from L_1 to L_2, the following holds:

$$\frac{\begin{array}{c}\forall i \in \{1 \dots n\}.\ \mathcal{T}(\mathtt{s}_i\mathtt{.l1}) = \mathtt{t}_i\mathtt{.l2} \\ s = \mathrm{load}(\mathtt{s}_1\mathtt{.l1} \circ \dots \circ \mathtt{s}_n\mathtt{.l1}) \qquad t = \mathrm{load}(\mathtt{t}_1\mathtt{.l2} \circ \dots \circ \mathtt{t}_n\mathtt{.l2})\end{array}}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

As before, these per-pass correctness results can be transitively composed to immediately conclude end-to-end correctness of $\mathcal{C}$.

So how do we prove this Level A per-pass correctness condition? Assuming that we have already proven whole-program per-pass correctness and are trying to port the proof over, there are two cases.

Verifying Per-Pass Correctness for Trivial Case Many compiler passes are inherently compositional, transforming the code of each module independently, *i.e.*, in a way that is agnostic to the presence of other modules. Put another way, such compiler passes commute with linking:

$$\mathcal{T}(\mathtt{s}_1\mathtt{.l1}) \circ \dots \circ \mathcal{T}(\mathtt{s}_n\mathtt{.l1}) = \mathcal{T}(\mathtt{s}_1\mathtt{.l1} \circ \dots \circ \mathtt{s}_n\mathtt{.l1})$$

If this commutativity property holds for a pass $\mathcal{T}$, then Level A per-pass correctness becomes a trivial corollary of whole-program per-pass correctness, where we instantiate the `s.l1` from Section 2.1 with $\mathtt{s}_1\mathtt{.l1} \circ \dots \circ \mathtt{s}_n\mathtt{.l1}$. In verifying Level A correctness for CompCert 2.4, we found that 13 of its 19 passes fell into this trivial case.

Verifying Per-Pass Correctness for Non-trivial Case If the trivial commutativity argument does not apply, then there is some new work to do to port a proof of whole-program per-pass correctness to Level A per-pass correctness.

However, at least for CompCert, we found it very easy to perform this adaptation. Why? First of all, since Level A correctness assumes that all modules in the program are transformed in the same way, we can essentially reuse the simulation relation R for pass $\mathcal{T}$ that was used in the original CompCert verification.

We do, however, have to worry about the soundness of the program analyses that the compiler performs, because the correctness of the compiler rests to a large extent on the correctness of these analyses. To prove Level A correctness of these analyses, we must prove that they remain sound even when they only have access to a single module in the program rather than the whole program. Intuitively, this should follow easily if: (1) the analyses have been proven sound under the assumption that they are fed the whole program (CompCert has done that already), *and* (2) the analyses are *monotone*, meaning that they only become *more conservative* when given access to a smaller fragment of the program (as happens with separate compilation).

In adapting CompCert to Level A correctness, the main work was therefore in verifying that its program analyses were indeed monotone. This was largely straightforward, with one exception: the "value analysis" employed by several optimizations was not monotone. It made an assumption about variables declared as `extern const`, which was valid for whole-program compilation, but not in the presence of separate compilation. As we explain in detail in Section 13, this manifested itself as a bug in constant propagation when linking separately-compiled files. After we fixed this bug in value analysis, monotonicity became straightforward to show, and thus so did Level A correctness (for the remaining 6 passes that did not fall into the trivial case).

12.2 Compositional Correctness Level B

End-to-End Correctness The CompCert compiler performs several key optimizations at the level of its RTL intermediate language (*i.e.*, they are transformations from RTL to RTL). For Level B, we would like to strengthen Level A correctness by allowing each source module $\mathtt{s}_i$ to be compiled by a *different* compiler $\mathcal{C}_i$. However, the differences we permit between the $\mathcal{C}_i$'s are restricted: they may differ only in which optimization passes they apply at the RTL level. Given this restriction on the $\mathcal{C}_i$'s, the Level B correctness statement is the same as the Level A correctness statement except for the

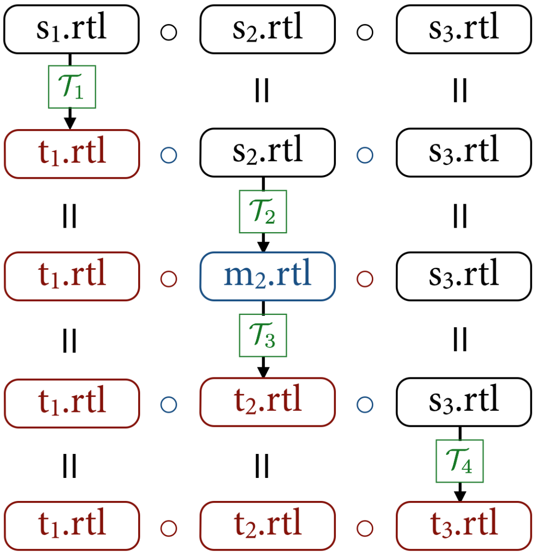


Figure III.2 Proving Level B correctness for RTL passes

replacement of $\mathcal{C}$ with $\mathcal{C}_i$:

$$\frac{\forall i \in \{1 \ldots n\}.\ \mathcal{C}_i(\mathtt{s}_i.\mathtt{c}) = \mathtt{t}_i.\mathtt{asm} \qquad s = \mathrm{load}(\mathtt{s}_1.\mathtt{c} \circ \ldots \circ \mathtt{s}_n.\mathtt{c}) \qquad t = \mathrm{load}(\mathtt{t}_1.\mathtt{asm} \circ \ldots \circ \mathtt{t}_n.\mathtt{asm})}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

Per-Pass Correctness To verify the above correctness statement, we yet again want to reduce it to verifying the individual passes of $\mathcal{C}$. For all passes besides the RTL-level optimizations, we can verify per-pass correctness exactly as in Level A, since all the $\mathcal{C}_i$'s must perform these same passes in the same order. However, for the RTL optimizations, we must do something different because at the RTL level the various $\mathcal{C}_i$'s do not all march in lock step.

The key idea for handling the RTL optimizations, as illustrated in Figure III.2 where each module is compiled with different optimization passes, is to pad the $\mathcal{C}_i$'s with extra dummy identity passes (which do not affect their end-to-end functionality) so that, whenever one compiler is performing an RTL optimization pass, the other compilers will for that step perform an identity transformation. Thus, we first verify the RTL passes of $\mathcal{C}_1$ in parallel with identity passes for the other compilers, then verify the RTL passes of $\mathcal{C}_2$ in parallel with identity passes for the other compilers, and so on. For this to work, the Level B per-pass correctness statement (for optimization passes $\mathcal{T}$ from RTL to RTL) must be updated as follows:

$$\frac{\begin{array}{c}\mathcal{T}(\texttt{s.rtl}) = \texttt{t.rtl} \\ s = \mathrm{load}(\texttt{u}_1\texttt{.rtl} \circ \ldots \circ \texttt{u}_m\texttt{.rtl} \circ \texttt{s.rtl} \circ \texttt{v}_1\texttt{.rtl} \circ \ldots \circ \texttt{v}_n\texttt{.rtl}) \\ t = \mathrm{load}(\texttt{u}_1\texttt{.rtl} \circ \ldots \circ \texttt{u}_m\texttt{.rtl} \circ \texttt{t.rtl} \circ \texttt{v}_1\texttt{.rtl} \circ \ldots \circ \texttt{v}_n\texttt{.rtl})\end{array}}{\mathrm{Behav}(s) \supseteq \mathrm{Behav}(t)}$$

One can view this notion of per-pass correctness as being essentially a form of *contextual refinement*: the output of $\mathcal{T}$ must refine its input when linked with a context consisting of some arbitrary other RTL modules (the $\texttt{u}_i$'s and $\texttt{v}_j$'s). If we can prove this, it should be clear from Figure III.2 how the per-pass proofs link up transitively.

Verifying Per-Pass Correctness So how do we prove this contextual refinement? Unlike for Level A, we cannot simply reuse the existing simulation R from the whole-program per-pass correctness proof for $\mathcal{T}$, because R is not necessarily reflexive and thus does not necessarily relate the execution of code from $\texttt{u}_i\texttt{.rtl}$ (or $\texttt{v}_j\texttt{.rtl}$) with itself. Instead, we must use an amended simulation R', which accounts for two possibilities: either we are executing code from `s` on one side of the simulation and code from `t` on the other, in which case the proof that R' is indeed a simulation proceeds essentially as did the proof that R was a simulation; or we are executing code from $\texttt{u}_i\texttt{.rtl}$ (or $\texttt{v}_j\texttt{.rtl}$), in which case both sides of the simulation are executing exactly the same RTL instructions.

In principle, the latter case could involve serious new proof effort. However, at least for CompCert, we found that in fact the substance of this new part of the proof was hiding in plain sight within the original CompCert verification! The reason, intuitively, is that RTL-to-RTL optimization passes are rarely unconditional transformations: their output typically only differs from their input *when certain conditions (e.g. determined by a static analysis) hold*, and since these conditions do not always hold, these passes may end up leaving any given input instruction unchanged. To account for this possibility, the original CompCert verification must therefore already prove that arbitrary RTL instructions simulate themselves. Consequently, in porting CompCert 2.4 to Level B compositional correctness, we were able to simply extract and compose (essentially, copy-and-paste) these micro-simulation proofs into the simulation proof for the latter part of R'.

Note: it is important to note that our idea works at porting CompCert's verification using forward simulations (when convenient), as well as that using backward simulations, to Level A and B compositional correctness.

13 Adapting Constant Propagation to Separate Compilation

In this section, we explain how to adapt the CompCert proof of constant propagation to support separate compilation. Before doing so, let us briefly explain syntactic linking in some more detail since it is central to the compositional correctness results we prove.

Syntactic linking merges global declarations for each identifier. The linker must check if the declarations meet the following conditions:

- The declarations have the same type. They should be either (1) function declarations or definitions of the same signature, or (2) variable declarations or definitions of the same type.
- At most one of the declarations is a definition. If there is a single definition, then that is the result of the linking; otherwise, everything is necessarily the same declaration, and that is the result of the linking.

We have generically defined syntactic linking for all the languages used in CompCert, as it does not depend on specific language features.

13.1 Verifying Compositional Correctness Level A

Adapting the Simulation Relation Definition To verify compositional correctness Level A, we will—as in the original CompCert proof—construct a simulation relation R that relates the initial states of the source and target programs. The difference is that the source program consists of multiple files, each of which is separately compiled.

$$\frac{\mathit{prg} = \mathtt{s}_1 \circ \ldots \circ \mathtt{s}_n \qquad \mathit{prg}' = \mathcal{T}_{\mathrm{cp}}(\mathtt{s}_n) \circ \ldots \circ \mathcal{T}_{\mathrm{cp}}(\mathtt{s}_n)}{\exists R.\ \text{simulation } R \wedge (\mathit{load}(\mathit{prg}), \mathit{load}(\mathit{prg}')) \in R}$$

Therefore, for each function definition (fd) in the source program (prg), the corresponding function definition (fd') in the target program (prg') is no longer obtained by $\mathrm{transfun}(\mathit{prg}, \mathit{fd})$, but rather by $\mathrm{transfun}(\mathtt{s}_i, \mathit{fd})$ for some subprogram $\mathtt{s}_i$ of prg. Moreover, the value analysis run as part of transfun also gets a subprogram of prg as its first argument.

Consequently, the simulation relation we use for the proof of soundness has to change. The main change is, naturally, in the definition of $\sim_{\mathrm{fdef}}$. Two function definitions are now related if the second can be obtained by transforming the first in the context of a subprogram of *prg*.

$$prg \vdash fd \sim_{\text{fdef}} fd' \overset{\text{def}}{=} \exists sprg \subseteq prg.\ fd' = \text{transfun}(sprg, fd)$$

where $sprg \subseteq prg$ iff $\exists sprg'.\ sprg \circ sprg' = prg$.[1]

The second change is to decouple the two uses of *prg* in sound-state, changing its signature so that it takes three arguments: two programs *prg* and *sprg*, and a state *s*. The first program, *prg*, corresponds to the full program and is used to calculate the global environment, whereas the second program, *sprg*, is a subprogram of the first one and is used to perform the global analysis (*i.e.*, to detect which variables are constant).

We then define a wrapper predicate sound-state′ as follows:

$$\text{sound-state}'(prg, s) \overset{\text{def}}{=} \forall sprg \subseteq prg.\ \text{sound-state}(prg, sprg, s)$$

and change R to use sound-state′ instead of sound-state.

Adapting the Proof of Value Analysis There are multiple proofs that require adaptation. First, we have to prove that value analysis is correct with respect to our stronger invariant. That is, we have to show that sound-state′ holds of the initial state of a loaded program and that it is preserved by execution steps.

The latter requirement is actually trivial to show and requires only very minor changes to the CompCert proof script. The reason for this, informally, is that the uses of the *prg* and *sprg* parameters in the revised sound-state invariant are really decoupled and that the preservation proof never depends on them being the same.

The former requirement, however, requires some more work: not only should $\text{sound-state}(prg, prg, \mathsf{load}(prg))$ hold for all programs *prg*, but rather $\text{sound-state}(prg, sprg, \mathsf{load}(prg))$ should hold for all programs *prg* and all subprograms $sprg \subseteq prg$. In essence, to satisfy this stronger statement, the additional requirement that we have to show is that value analysis is monotone with respect to linking. That is, for each global variable x, we prove

$$sprg \subseteq prg \implies \text{abs-val}(sprg, \mathsf{x}) \sqsupseteq \text{abs-val}(prg, \mathsf{x})$$

where $\text{abs-val}(prg, \mathsf{x})$ is the abstract value of x computed by the global analysis on *prg*. In other words, running the analysis on a larger program may only give results that are at least as precise.

[1] In the Coq development, we define $sprg \subseteq prg$ in an equivalent but more direct, semantic way, rather than relying on syntactic linking ($\circ$). This enables us to avoid having to prove associativity and commutativity of linking.

```c
// a.c
#include <stdio.h>
int x;
extern int* const xptr;
int main() {
  x = 1;
  *xptr = 0;
  // expected: 0, actual: 2
  printf("%d\n",x+x);
  return 0;
}
```

```c
// b.c
#include <stdio.h>
extern int x;
int* const xptr = &x;
```

Figure III.3 A bug due to CompCert 2.4 value analysis

Somewhat surprisingly, (the global part of) the value analysis in CompCert 2.4 does not satisfy this monotonicity requirement because of its treatment of variables declared as both `extern` and `const`. Figure III.3 presents two C files that, when compiled separately and linked together, expose the bug. Since the global variable `xptr` is declared using the `const` qualifier, the global part of CompCert 2.4's value analysis assumes that it is uninitialized and therefore assigns it the abstract value $\bot$. As a result, the value analysis deems that `xptr` cannot possibly alias with `x`. At the `printf` statement, it hence deduces that `x` = 1, and constant propagation "optimizes" away the summation `x+x` to the constant 2, which gets printed. This analysis, however, is unsound. In particular, the assumption that `xptr` is uninitialized is invalid in the context of multiple separately compiled files: since `xptr` is also declared as `extern`, another file (`b.c`) can provide a definition that initializes it. And indeed, since the definition of `xptr` in `b.c` causes `x` and `xptr` to alias, the correct result is 0, not 2.

Restoring soundness of the value analysis is straightforward: one simple if rather crude fix (which has been adopted by CompCert 2.5 since we reported the bug) is just to ignore the `const` modifier on `extern` declarations. Having done that, it is easy to show that the analysis is monotone with respect to program linking and that therefore the initial state of loading a program satisfies the stronger invariant $\text{sound-state}'(\mathit{prg}, \mathsf{load}(\mathit{prg}))$.

Adapting the Proof of Constant Propagation Showing that constant propagation is sound requires only very little additional work.

The only important difference is in the treatment of global environments that index the $\hookrightarrow$ relation. Generally, these environments are obtained by the respective programs

($ge = \mathit{get\text{-}genv}(prg)$ and $ge' = \mathit{get\text{-}genv}(prg')$).

The original CompCert 2.4 proof used the fact that $prg' = \mathcal{T}_{\mathrm{cp}}(prg)$ to establish a relationship between ge and ge'. It proved two basic properties relating the two global environments: (1) that they map each global variable name to the same block identifier, and (2) that if ge maps a block identifier to a function signature or definition fds, then ge' maps it to $\mathrm{transfun}(prg, fds)$, where transfun applied to a function signature returns the same signature. These lemmas were then used in the proof that R is a simulation relation.

Since, now, the relationship between prg and prg' is more involved, we have to update the proof of the first lemma, which is rather straightforward, as well as the statement and proofs of the second lemma. For the second lemma, we now assert that if ge maps a block identifier to a function signature or definition fds, then ge' maps it to $\mathrm{transfun}(sprg, fds)$ for *some* subprogram $sprg \subseteq prg$. Besides this change, the proof that now (the new definition of) R is a simulation relation is basically unchanged. The few lines of the proof script that required editing were those invoking the lemmas about the relationship between the global environments.

13.2 Verifying Compositional Correctness Level B

Adapting the Simulation Relation Definition We move on to verifying the second level of compositional correctness, that of composition with the same compiler modulo optimization flags. As we have explained in Section 12.2, for every optional optimization pass, we need to show that linking it against the identity compiler is sound. Since constant propagation is one of the optional optimization passes, we have to prove the following:

$$\frac{\begin{array}{c} prg = \mathtt{u}_1 \circ \ldots \circ \mathtt{u}_m \circ \mathtt{s} \circ \mathtt{v}_1 \circ \ldots \circ \mathtt{v}_n \\ prg' = \mathtt{u}_1 \circ \ldots \circ \mathtt{u}_m \circ \mathcal{T}_{\mathrm{cp}}(\mathtt{s}) \circ \mathtt{v}_1 \circ \ldots \circ \mathtt{v}_n \end{array}}{\exists R.\ \mathrm{simulation}\ R \wedge (\mathit{load}(prg), \mathit{load}(prg')) \in R}$$

In the scenario above, the corresponding target function definition fd' of a source function definition fd is either syntactically identical to fd if it belongs to one of the $\mathtt{u}_i/\mathtt{v}_j$ files, or has been obtained by optimizing fd if it belongs to the $\mathtt{s}$ file. To account for the change, we should therefore redefine the $\sim_{\mathrm{fdef}}$ relation as follows:

$$\begin{array}{l} prg \vdash fd \sim_{\mathrm{fdef}} fd' \overset{\mathrm{def}}{=} \\ \qquad\qquad fd' = fd \vee (\exists sprg \subseteq prg.\ fd' = \mathrm{transfun}(sprg, fd)) \end{array}$$

This is the only change needed in the simulation relation.

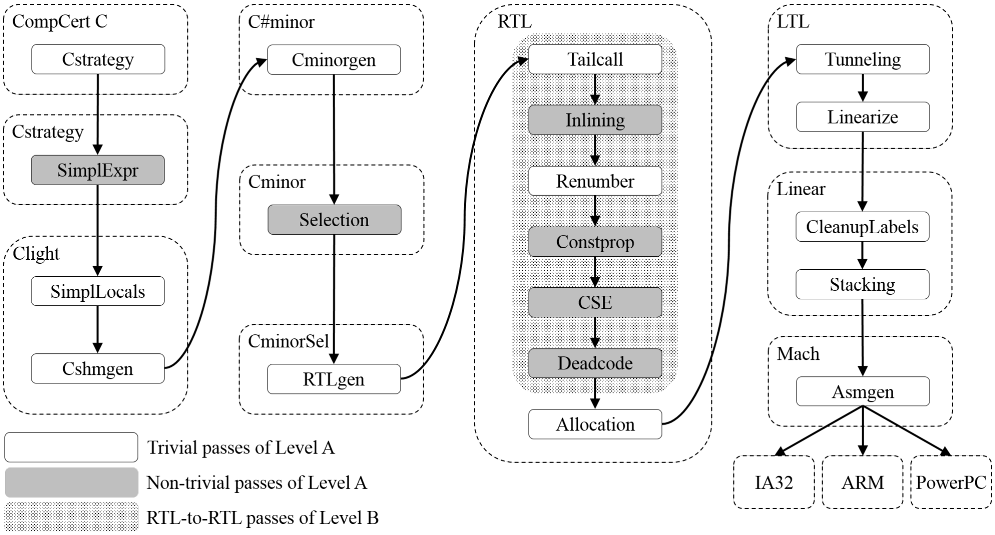


Figure III.4 Classification of optimization passes in CompCert

Adapting the Proof of Constant Propagation To avoid changing the proof script of the main lemma showing that R is in fact a simulation, we prove a helper lemma (*transf_step_correct_identical*) saying that, given two matching instruction states with $fd' = fd$, when the source state takes a step, the target state can also take a step and reach a matching state. As explained in Section 12.2, the content of the proof of this helper lemma is already present in the existing simulation proof, just not in one place, so it simply needs to be extracted and consolidated. We then adapt the proof of the main lemma (showing R is a simulation) so that it performs a case split on whether $fd = fd'$ or not, and correspondingly either invokes our helper lemma or uses the same proof script as for Level A.

14 Adapting the Other Passes to Separate Compilation

In the previous section, we looked at the specific example of constant propagation in detail and explained how we adapted CompCert's proof of that pass to Level A and B notions of compositional correctness. In this section, we discuss some details of other CompCert passes for which adapting to Level A and B correctness required some interesting (but still not much) work.

Figure III.4 shows which verification technique we apply to each optimization pass of CompCert. For Level A, we apply the trivial technique based on commutativity with linking to 13 passes and the non-trivial technique to 6 passes. For Level B, we apply our technique to all 6 RTL-to-RTL passes.

14.1 RTL-Level Optimizations that Rely on Value Analysis

Three RTL-level optimizations rely on the value analysis: constant propagation, common subexpression elimination (CSE), and deadcode elimination (DCE). These passes are inter-procedural solely because they rely on the value analysis. Thus, the porting of the proofs of CSE and DCE to support compositional correctness proceeded analogously to the porting of the constant propagation pass.

14.2 Selection

The selection pass "recognizes opportunities for using combined arithmetic and logical operations and addressing modes offered by the target processor." [22]. The pass is mostly intra-procedural, except for the following two transformations on function calls:

Recognizing Immediate Calls The selection pass transforms an indirect call via a function pointer expression, say e_p, into an immediate call, if it can determine that the expression e_p always evaluates to the pointer to an internal function. The pass uses a simple analysis `classify_call`(prg, e_p) for determining this. For separate compilation, we proved the monotonicity of the analysis: the result `classify_call`($sprg$, e_p) for a subprogram $sprg \subseteq prg$ is sound w.r.t. the whole program prg.

64-bit Integer Operations into Library Calls The selection pass transforms some 64-bit integer operations into calls to library helper functions. For example, `(long long) f`, a cast from `float` to `long long`, is transformed into a call `__int64_dtos(f)` to the corresponding helper function. For this transformation to be valid, the pass should ensure that the helper function (*e.g.*, `__int64_dtos`) is declared as an external function in the source program's global environment. CompCert has a designated checker `check_helpers`(prg) to ensure this property.

For separate compilation, we proved that the checker for helper functions is monotone w.r.t. linking: `check_helpers`($prg1$) and `check_helpers`($prg2$) implies `check_helpers`($prg1 \circ prg2$) for all programs $prg1$ and $prg2$. The proof is a little bit involved, as linking reorders global declarations, and the corresponding logical blocks for the helper functions in the global environments may vary.

Compiler Bug We Found The original CompCert 2.4 used a function `get_helpers` instead of `check_helpers`. We found and reported a compiler bug related to `get_helpers`, which was subsequently confirmed.

We found the bug in the course of proving monotonicity regarding `get_helpers`. This function is directly implemented in OCaml and its property is axiomatized in Coq as follows:

```
Axiom get_helpers_correct:
  forall ge hf, get_helpers ge = OK hf ->
  i64_helpers_correct ge hf.
```

The problem was that this axiom is not strong enough to prove monotonicity, and even worse, not true for the OCaml implementation of `get_helpers`. One of the properties this axiom postulates is that helper functions like `__int64_dtos` are only declared but not defined in the source program. However, `get_helpers` does not check it at compile time!

Here is an example that exposes the bug:

```
#include <stdio.h>
long long __i64_dtos(float t) {
  return 3;
}
int main() {
  printf(“%lld\n”, (long long) 5.0f); // expected: 5, actual: 3
  return 0;
}
```

Here the cast `(long long) 5.0f` is converted to a call to the library helper function `__i64_dtos(5.0f)` by the selection pass. However, we successfully hijack the function call by overwriting the function `__i64_dtos`, which results in printing 3 instead of the correct result 5. Now, strictly speaking, due to the hijacking of a reserved identifier, this example has undefined semantics according to the C99 standard, so CompCert's behavior here is technically legal. But the dependence on an invalid axiom is clearly a bug.

After we reported this bug, it was fixed in the development branch of CompCert (and subsequently CompCert 2.5). In this fix, which we backported to CompCert 2.4 using `git-cherry-pick`, the OCaml `get_helpers` function is replaced by the aforementioned `check_helpers`, which is implemented and verified directly in Coq, thereby avoiding the need for an invalid axiom.

14.3 Inlining

The inlining pass is inherently inter-procedural, as it replaces a call to a simple function with the body of the function. In the pass, the selector `funenv_program`(prg) chooses internal function definitions of prg that are worth inlining in other functions. For the inlining pass to be valid, the pass should ensure that function definitions in `funenv_program`(prg) are indeed defined in the global environment $\textit{get-genv}(prg)$.

For separate compilation, we proved that the global environment initialization is monotone for function definitions: if a function definition, say fd, is defined in $\textit{get-genv}(sprg)$ and $sprg \subseteq prg$, then fd is also defined in $\textit{get-genv}(prg)$. The proof is a little bit involved for the same reason as for `check_helpers` in the selection pass: linking reorders global declarations and the corresponding logical blocks in the global environments.

14.4 SimplExpr

The SimplExpr pass is essentially intra-procedural since just "side effects are pulled out of CompCert C expressions" [22]. However, it does not commute with linking because a single counter is used globally to generate temporary variable names for all function definitions.

Updating the existing proof was easy because the original simulation relation R does not bake in the specific way temporary variable names are generated. Thus, we did not need to change the simulation relation R and the simulation proof at all. We just needed to update the proof that the initial states after loading satisfy the relation R even in the presence of separate compilation, which was straightforward.

15 Results

We applied our Level A and B techniques to CompCert 2.4 with three patches applied (two bug fixes and RTL-level optimization flags). In the former, we prove the behavioral refinement result between the source C program obtained by syntactically linking several C files and the target assembly program obtained by syntactically linking the results of compiling source files with the *same* optimization flag. In the latter, we prove the same behavioral refinement result even when each source file is compiled with a different optimization flag.

	CompCert	LevelA Correctness			LevelB Correctness		
	(LOC)	Rm	AddD	AddN	Rm	AddD	AddN
Total	206702	318 (0.2%)	465 (0.2%)	3392 (1.6%)	372 (0.2%)	1439 (0.7%)	4845 (2.3%)
Compiler & Verification	88451	318 (0.4%)	465 (0.5%)	1153 (1.3%)	372 (0.4%)	1439 (1.6%)	1726 (2.0%)
driver/Compiler.v	367	6 (1.6%)	6 (1.6%)		9 (2.5%)	9 (2.5%)	
../ValueAnalysis.v	1825	16 (0.9%)	33 (1.8%)	86 (4.7%)	16 (0.9%)	33 (1.8%)	86 (4.7%)
../Cstrategy.v	3070						
../SimplExpr(proof|spec).v	3383	14 (0.4%)	10 (0.3%)	68 (2.0%)	14 (0.4%)	10 (0.3%)	68 (2.0%)
../SimplLocalsproof.v	2251			19 (0.8%)			19 (0.8%)
../Cshmgenproof.v	1515			21 (1.4%)			21 (1.4%)
../Cminorgenproof.v	2256			11 (0.5%)			11 (0.5%)
../Select*proof.v	2686	86 (3.2%)	117 (4.4%)	203 (7.6%)	86 (3.2%)	117 (4.4%)	203 (7.6%)
../RTLgen(proof|spec).v	2781			12 (0.4%)			12 (0.4%)
../Tailcallproof.v	627			8 (1.3%)	21 (3.3%)	186 (29.7%)	37 (5.9%)
../Inlining(proof|spec).v	1979	59 (3.0%)	98 (5.0%)	50 (2.5%)	60 (3.0%)	323 (16.3%)	64 (3.2%)
../Renumberproof.v	267				30 (11.2%)	126 (47.2%)	35 (13.1%)
../Constpropproof.v	644	50 (7.8%)	68 (10.6%)	26 (4.0%)	52 (8.1%)	180 (28.0%)	36 (5.6%)
../CSEproof.v	1238	29 (2.3%)	56 (4.5%)	34 (2.7%)	30 (2.4%)	146 (11.8%)	45 (3.6%)
../Deadcodeproof.v	1024	58 (5.7%)	77 (7.5%)	34 (3.3%)	54 (5.3%)	171 (16.7%)	45 (4.4%)
../Allocproof.v	2219			14 (0.6%)			14 (0.6%)
../Tunnelingproof.v	417						
../Linearizeproof.v	750			8 (1.1%)			8 (1.1%)
../CleanupLabelsproof.v	372						
../Stackingproof.v	2894			10 (0.3%)			10 (0.3%)
../Asmgenproof0proof.v	867						
arm/*proof*.v	4046						
ia32/*proof*.v	3683						
powerpc/*proof*.v	3912						
others in cfrontend, backend, driver, arm, ia32, powerpc	43378			549 (1.3%)		138 (0.3%)	1012 (2.3%)
Metatheory	118251			2239 (1.9%)			3119 (2.6%)

Rm: LOC removed from CompCert **AddD**: LOC added but derived from CompCert **AddN**: LOC newly added

%: ratio to the LOC of CompCert ▨ shaded cell : interesting changes

Figure III.5 Changes to lines of codes for LevelA and LevelB correctness

In the table in Figure III.5, we summarize the changes we made in number of lines of Coq. For these statistics, we first split the development of CompCert into two categories: (1) Compiler & Verification; and (2) Metatheory (*i.e.*, everything else). The former includes all Coq files in the directories `cfrontend`, `backend`, `driver`, `arm`, `ia32`, and `powerpc`; and the latter includes all other files.

We calculated the statistics fully automatically using the Unix `diff` command. The column **Rm** shows the number of lines of code (LOC) removed from the original CompCert code reported by the diff command, and the columns **AddD** and **AddN** together show the number of LOC added. Here, **AddD** counts the LOC that are derived from the original CompCert code including copy-pasted-and-modified code and **AddN** the LOC that we newly proved. We syntactically marked the newly proved code, so that we can automatically distinguish **AddD** and **AddN**.

The shaded part in the table denotes interesting changes we made. In Level A, the shaded part is mainly due to proving various monotonicity properties. In Level B, the shaded part is mainly due to copy-paste-and-modifying the original proof of simulation. Examples of (what we consider) uninteresting changes, which are not shaded, include (a) proving straightforward "wheel-greasing" lemmas that merely serve to streamline the proof effort, and (b) updating automatically generated hypothesis names appropriately (*e.g.*, changing `apply H1` to `apply H2`).

Our verification supports separate compilation to all three target assembly languages of CompCert—PowerPC, ARM, and IA32—with very few changes made to the original proofs. In the whole development, for Level A, we modified only 0.2% of the existing code (**Rm**) and introduced an additional 1.6% (**AddN**+**AddD**-**Rm**); and for Level B, we modified only 0.2% of the existing code (**Rm**) and introduced an additional 2.8% (**AddN**+**AddD**-**Rm**). We spent less than two person-months in total for the whole development, much of which was spent understanding the existing CompCert development.

16 Discussion

16.1 Related Work

Compositional CompCert The most closely related work to ours is Stewart *et. al.* 's Compositional CompCert [76], which establishes compositional correctness for a significant subset of CompCert 2.1. Their approach builds on their previous work on *interaction semantics* [16], which defines linking between modules in a (somewhat) language-independent way. (The languages in question must share a common memory model, as is the case for C, assembly, and all the intermediate languages of CompCert.) Essentially, interaction semantics enables Compositional CompCert to reduce the compiler verification problem to one of contextual refinement. The output of the compiler is proven to refine the source under an arbitrary "semantic context", which may consist of a linking of C and assembly modules.

On the one hand, Compositional CompCert is targeting a more ambitious goal than separate compilation. Their approach inherently supports the possibility of linking results of multiple different compilers, as well as the ability to link C modules with hand-coded assembly modules, both of which are beyond the scope of our techniques.

On the other hand, as we explained in the introduction, the modesty of our goal is quite deliberate—it enables our separate compilation verification to use a considerably more lightweight approach than they do. For example, they report that their porting of CompCert passes to verify compositional correctness took approximately 10 person-months and led to more than a doubling in the size of each pass. In contrast, our porting took less than 2 person-months, and even if we look at just the passes alone (ignoring the metatheory), they are on average less than 2% (for Level A) or 4% (for Level B) larger than the original CompCert passes. The new metatheory backing up our proof technique is also smaller than the corresponding new metatheory of Compositional CompCert by roughly a factor of 7.

One reason for this, we believe, is that the *structured simulations* employed in the Compositional CompCert proof require all passes in the compiler to be verified using a common "memory-injection" invariant. In contrast, in our verification, we were able to essentially reuse the invariants from the original CompCert proof, which are different for different passes. Another potential reason concerns the treatment of inter-module vs. external function calls. In CompCert, external functions are assumed to satisfy a number of axioms, which are used in the verification to establish that external function calls preserve the simulation relation in each pass. In Compositional CompCert, inter-module function calls are treated the same as external function calls, and as a result Stewart *et. al.* 's verification must additionally establish that functions compiled by the compiler satisfy the external function axioms. In contrast, in our verification, we reduce the problem of verifying separate compilation to that of verifying correctness of compilation for a whole (multi-module) program. Thus, for us, inter-module function calls are *not* treated as external calls—they merely shift control to another part of the program—and there is no need for us to prove that CompCert-compiled functions satisfy the external function axioms.

There are two other points of difference worth mentioning. First, we have ported over the entire CompCert 2.4 compiler, from C to assembly (including the x86, Power, and ARM backends), whereas Compositional CompCert omitted the front-end of CompCert 2.1 (from C to Clight), along with three of its RTL-level optimizations (CSE, constant propagation, and inlining). Second, due to its use of interaction semantics, Compositional CompCert employs a bespoke "semantic" notion of linking (even for linking assembly files), which has not yet been related to the standard notion of syntactic linking (see Section 13) that we employ in our verification. Syntactic linking corresponds much more closely to the physical linking of machine code implemented by the `gcc` linker that CompCert uses by default. Proving this is outside the scope of our verification effort, however, since CompCert only verifies correctness of compilation down to the assembly level, not to the machine-code level where linking is actually performed.

CompCertX Concurrently with Stewart *et. al.* 's work, Ramananandro *et. al.* [71] developed a different approach to compositional correctness for CompCert. While similar in many ways to the approach of Stewart *et. al.* , the *compositional semantics* of Ramananandro *et. al.* defines linking so that the linking of assembly-level modules boils down to syntactic linking (essentially, concatenation), as it does in our verification. On the other hand, they have only used their approach to compositionally verify a few passes of CompCert.

Also concurrently with the above work, Gu *et. al.* [30] have developed CompCertX, a compositional adaptation of CompCert specifically targeted for use in the compositional verification of OS kernels. CompCertX supports linking of compiled C code with hand-written assembly code, and ports over all passes of CompCert, but the source and target of the compiler are different from CompCert's. In particular, the ClightX source language of CompCertX does not generally allow functions to modify other functions' stack frames and thus does not support stack-allocated data structures. Although this restriction is not problematic for their particular application to OS kernel verification, it means that, as the authors themselves note, "CompCertX can not be regarded as a full featured separate compiler for CompCert."

Multi-lanugage Semantics There have been several approaches proposed for compositional correctness of compilers other than CompCert as well, although these all involve *de novo* verifications rather than ports of existing whole-program compiler verifications.

Perconti and Ahmed [66] present an approach to compositional compiler correctness for ML-like languages. They use multi-language semantics to combine all the languages of a compiler into one joint language, with wrapping operations to coerce values in one language to values of the appropriate type in the other languages. Like Compositional CompCert, their approach recasts the compiler verification problem as a contextual refinement problem, except that they model contexts syntactically rather than semantically and use *logical relations* as a proof technique for establishing contextual refinement rather than structured simulations. It is difficult to gauge how well this approach scales as a practical compiler verification method because it has not yet been mechanized or applied to a full-blown compiler.

Cito Wang *et. al.* [79] develop a compositional verification framework for a compiler for Cito, a simple C-like language [79]. Their approach is quite different from the others in that it characterizes the compiler verification problem in terms of Hoare-style specifications of assembly code. Currently, the approach is limited in its ability to talk about preservation of termination-sensitive properties, and as its verification statement is so different from the traditional end-to-end behavioral refinement result established by a compiler like CompCert, it is not clear how Wang *et. al.* 's method could reuse existing CompCert-style verifications.

Parametric Inter-Language Simulation Most recently, Neis *et. al.* [61] present *parametric inter-language simulations (PILS)*, which they use to compositionally verify Pilsner, a compiler for an ML-like core language, in Coq. PILS build on earlier work by Hur *et. al.* on logical relations [15, 31] and parametric bisimulations (aka relation transition systems) [32, 34]. Pilsner supports a very strong compositional correctness statement, but it also required a major verification effort, involving several person-years of work and 55K lines of Coq.

16.2 Generality of Our Techniques

In this paper, we have presented several simple techniques for establishing Level A and Level B compositional correctness, and demonstrated the feasibility and effectiveness of these techniques by porting a major landmark compiler verification (CompCert 2.4) to support compositional correctness without much difficulty at all. We hope the almost embarrassingly simple nature of these techniques will encourage future compiler verifiers to consider proving at least a restricted form of compositional correctness for their compilers from the start.

One may wonder, however, how general our techniques are. Are they dependent on particular aspects of CompCert 2.4? Can they be applied to other verified compilers?

For C or for other languages? Given the landscape of compiler verification, dotted as it is with unique and majestic mountains, it is difficult to give sweepingly general answers to these questions. But we can say the following.

We believe our techniques should be applicable to the most recent version of CompCert (2.5), but a necessary first step is to determine the appropriate notion of syntactic linking. CompCert 2.5 introduces support for `static` variables (which in C means variables that are only locally visible within a single file). The presence of `static` variables means that the simple canonical definition of syntactic linking we have used no longer works and must be revisited. Assuming a reasonable definition can be found, as we expect, we do not foresee any problems adapting our techniques to handle it.

Regarding the application to compilers for other languages, we can only speculate, but we also do not foresee any fundamental problems. For instance, CakeML [49] is a verified compiler for a significant subset of Standard ML, implemented in HOL4. CakeML's end-to-end verification statement concerns the correctness of an x86 implementation of an interactive SML read-eval-print loop. In that sense, the verification is not exactly "whole-program" because new code can be compiled and added to a global database interactively. But it also does not support true separate compilation in the sense that our verification does because modules cannot be compiled independently of the other modules they depend on.

We believe in principle it should be possible to use our techniques to adapt CakeML to verify correctness of separate compilation, because CakeML is not an optimizing compiler and in particular does not perform any optimizations that depend fundamentally on the whole-program assumption. The key challenge will be figuring out how to define separate compilation and linking themselves. The latter may be especially interesting since CakeML (unlike CompCert) verifies correctness of compilation all the way down to x86-64 machine code, and thus linking will need to be defined at the machine-code level.

16.3 Impact

We verified separate compilation for the *full* CompCert compiler for the first time. In the course of doing so, we uncovered two compiler bugs—one of which is on separate compilation and the other is orthogonal to separate compilation—and our verification techniques are subsequently adopted in the official CompCert 2.7.

Chapter IV

Cast between Integers and Pointers

17 Introduction

Unrestricted manipulation of the representation of pointers via cast between pointers and integers is crucially used in low-level code. For example, the Linux kernel and the HotSpot Java virtual machine perform bitwise operations on pointers in maintaining page tables and live objects, respectively. For another example, the C++ standard library's hash function (`std::hash`) uses the pointer's bit representation as a key, *e.g.*, for indexing into a hash table. This is useful since taking a pointer is a cheap way to get a unique key.

Problem However, none of the existing proposals for C/C++ semantics fully supports manipulation of pointers as integer values *and* compiler optimizations at the same time. ISO C18 standard provides an integer type `uintptr_t` that may be legally cast to and from pointer types, it does not require anything of the resulting values [37, §7.20.1.4p1]. The concrete model straightforwardly supports bit-level pointer manipulation, but as we have seen in Section 2.2 and 3, this model invalidates many basic compiler optimizations such as constant propagation and dead allocation elimination. On the other hand, CompCert's logical model presented in Section 2.2 and 3 supports such compiler optimizations, but it does not support cast between pointers and integers—as well as many other low-level features in C—because the logical model represents pointers as

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
    xi = i;
  }
  auto p = (int *) xi;
  *p = 1;
  printf("%d\n", x); // expected: 1, actual: 0
}
```

Figure IV.1 A GCC bug in the presence of integer-pointer casts

pairs of an allocation block identifier and an offset within that block, which cannot be easily casted into and from an integer.

Designing a satisfactory semantics of integer-pointer casts is challenging. For example, such a semantics should point out which optimization(s)—each of which seems legit at first glance—is to blame in the LLVM miscompilation bug presented in Figure I.1.

Such a semantics should also guide compiler writers how to fix the GCC miscompilation bug we found, which is presented in Figure IV.1.[1] (At first, ignore the gray area.) In the program, the pointer to the local variable `x` is cast to an integer, `xi`, and then cast back to pointer, `p`. Thus `p` points to `x`, and after `*p` is assigned one, the value of `x` should be one. Now inserting the code in the gray area should not change the program's behavior, because the gray area is dead code: after the for loop, `i` equals to `xi`, and the conditional branch is not taken. But GCC miscompiles the program as follows:

1. Code motion optimization moves `xi = i` out of the conditional branch, because regardless of whether the branch is taken, `xi = i` should hold.

2. Alias analysis thinks that `p` is not a valid pointer, because it originates from `xi`, which equals to `i`, which in turn is obtained by just incrementing by one several times from zero.

3. Constant propagation optimization replaces `x` at the last line with zero, because

[1] We reported this miscompilation bug as a comment in `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752`, which is still open as of this writing.

x is initialized with zero and no intervening stores are writing to x. In particular, p is not aliased with &x, at least from the compiler's point of view, because p is not a valid pointer.

Our Solution In this chapter, we propose a C/C++ memory model that gives semantics to programs that manipulate the bit-level representation of pointers, and yet permits the same optimizations as logical models for code not using these low-level features. The key technical ingredient for making this work is combining the strengths of the concrete and logical memory models: now pointer values have two distinct representations, a concrete and a logical one, and they can be converted to each other. By default, a pointer is represented logically, and only when it is cast to an integer type, is the logical pointer value *concretized* to a concrete 32-bit integer (or 64-bit integer depending on the architecture). When an integer is cast back to a pointer value, it is mapped to the corresponding logical address.

With our hybrid model, we propose how to fix both the LLVM bug presented in Figure I.1 and the GCC bug presented in Figure IV.1, which is by turning off some controversial, too aggressive alias analysis. Furthermore, we generalized the existing compiler verification techniques for the logical memory model to also account for our hybrid model. Our model also supports all the reasoning principles designed for the logical model—*i.e.*, any sound reasoning about programs in the logical model also holds in our model—because our hybrid model conservatively extends the logical model: it gives semantics to strictly more programs than those supported by the logical model without changing their semantics.

To summarize, our contributions are:

- The first formal semantics that *fully* supports unrestricted manipulation of pointer values via integer-pointer casts and yet allows the standard compiler optimizations (Section 18).
- A proposal to fix to the LLVM bug presented in Figure I.1 and the GCC bug presented in Figure IV.1 (Section 19).
- Compiler verification techniques for proving semantic preservation under our semantics and their application to verify a number of standard optimizations in the presence of integer-pointer casts (Section 20).

Section 21 discusses the related work and the impact of our memory model. All the proofs reported in this chapter have been fully formalized in Coq and is available online [1].

18 Formal Semantics of Hybrid Model

Our model is simply a hybrid of the fully concrete model and the fully logical model. However, there are several issues with how to combine the two models to minimize their disadvantages. In this section, we introduce the hybrid model and discuss how we address the design issues at a high level.

All the optimization examples presented in this section are performed by `clang -O2`. Furthermore, unless specified otherwise, integer variables have type `uintptr_t`, and pointer variables have type `int*`. Recall that `uintptr_t` is an integer type that is able to hold a pointer value.

18.1 Hybrid of Concrete and Logical Blocks

Our hybrid model slightly generalizes the logical model to allow both concrete blocks (as in the concrete model) and logical blocks (as in the logical model). For this, we add one more attribute p to a logical block presented in Section 2.2, which is either *undefined* or a concrete address. The attribute p indicates whether the block is logical (when p is undefined) or it is a concrete block starting at the address p (when p is defined).

$$\begin{aligned}\text{Block} \overset{\text{def}}{=} \{\, (v, p, n, c) \mid\ & p \in \texttt{int32} \uplus \{\text{undef}\} \\ & \wedge v \in \{\texttt{valid}, \texttt{freed}\} \wedge n \in \mathbb{N} \wedge c \in \text{Val}^n \,\}\end{aligned}$$

We say that an address (l, i) is concrete when the block l is a concrete block starting at an address p. In this case, the address (l, i) can be cast to the integer $p + i$ and vice versa (see Section 18.8 for details).

As in the concrete model, the list of valid (*i.e.*, allocated) blocks with concrete addresses must be consistent: they should not include zero (*i.e.*, `nullptr`) or the maximum address, and their ranges should be disjoint. Logical blocks have no such requirement, since they are non-overlapping by construction.

In the subsequent subsections, we discuss several issues that arose during the design of our hybrid model and justify our solutions to the issues.

18.2 Combining Logical and Concrete Blocks

Our hybrid model allows both concrete and logical blocks to coexist, suffering from the disadvantages of both kinds of blocks: concrete ones do not provide exclusive ownership and logical ones do not allow casting to integers. Then why do we not develop a new notion of block that has the advantages of both concrete and logical blocks instead?

```
a = (a - b) + (2 * b - b);
q = (int *) a;
*q = 123;
```

→

```
q = (int *) a;
*q = 123;
```

Figure IV.2 Arithmetic optimization example I

Because such a new notion of block would not justify other important optimizations such as simplification of integer operations. For instance, consider a model in which some blocks have both concrete addresses and some extra permission information, so that we can tell when a block is exclusively owned. In such a model, we would like to know that we do not lose permission information when a pointer is cast to an integer, even if integer operations are performed on it (*e.g.*, `base64_encoding` a pointer and then `base64_decoding` it). However, this prevents the optimization presented in Figure IV.2. Suppose the variable `b` contains an integer with permission to access some valid block l, and `a` contains an integer without any permission that is equal to the concrete address of the block l. Then the source program successfully stores 123 into the block l because `q` has the relevant permission, whereas the target program fails because `q` does not have the permission.

Notice that this optimization is sound in our model. See Section 20.5 for how to verify it in our model.

18.3 Choosing Concrete Blocks

As discussed in Section 2.2, using concrete addresses for memory locations provides no guarantees of ownership, and thus prevents certain optimizations. In the worst case, one function could guess the concrete address of a supposedly-private resource of another function, and then forge a pointer to that address and modify it.

In order to maximize the range of optimizations that can be performed, our hybrid model assigns concrete addresses to only those blocks whose concrete addresses are requested via pointer-to-integer casts. This gives a simple semantics of pointer-to-integer casts in that (1) it clearly defines when a block should be made concrete, and (2) the semantics of integer operations is still independent from the memory in the presence of pointer-to-integer casts.

The reader may wonder if we can further maximize the optimization opportunity by making concrete only those blocks whose concrete addresses are really used in some operation. If we perform some computation with the value of a pointer that only makes sense when that value is an integer (*e.g.*, comparing it with an integer value) then the target of that pointer must have a concrete address. In all other cases, even if the ad-

```
void foo(uintptr_t a) {          foo(uintptr_t a) {
    a = a & 123;                     a = a & 123;
}                                }
...                          →   ...
a = (uintptr_t) p;               a = (uintptr_t) p;
foo(a);
bar();                           bar();
```

Figure IV.3 Dead code elimination example

dress of the block is taken, we could conceivably use a logical value and maintain the ownership guarantees of the logical model.

However, this approach has a serious problem: it does not justify some important integer optimizations, such as a dead code elimination presented in Figure IV.3. Suppose the pointer `p` contains a logical block l. In the source program, since its concrete address is used in the function `foo`, the block l must be given a concrete address. In the target, the read-only call to `foo` is optimized away, and the block l may not be given a concrete address. That is, the source may have more concrete blocks than the target. Thus, if `bar()` accesses an arbitrary concrete memory location, then that access might succeed in the source but fail in the target. Since a failure is possible in the target that did not exist in the source, the optimization has introduced new behavior, and is invalid.

Notice that this optimization is sound in our hybrid model, because it makes concrete all the blocks whose addresses are cast to integers, even if the cast integers are not used in any operation. See Section 20.5 for how to verify this example in our model. Furthermore, our model allows most of the optimizations in practice that would be performed in the minimally-concrete model presented above (see Section 18.7 for details).

18.4 Assigning Concrete Addresses

Once we know which blocks will need concrete addresses, we need to decide when during a program's execution to assign those addresses.

One approach would be to make such a decision as early as possible (*i.e.*, at allocation time). We allocate blocks as either logical or concrete, and cause concrete operations (namely integer casts) on logical blocks to raise *out-of-memory-type behavior*. (For its precise meaning, we refer the reader to Section 20.1.) Since it is difficult to determine whether a block will need a concrete address, we would need to choose the kind of block to allocate non-deterministically. However, this would add unintuitive failures to our model, effectively allowing out-of-memory-type behavior when the allocator chooses the wrong kind of block even when concrete blocks are available.

```
p = (int *) malloc(4);          p = (int *) malloc(4);
*p = 123;                       *p = 123;
bar();                    →     bar();
a = *p;                         a = *p;
hash_put(h, p, a);              hash_put(h, p, 123);
```

Figure IV.4 Ownership transfer example

Our solution is to instead allocate all blocks as logical blocks, and assign concrete addresses to logical blocks at casting time. This casting can result in out of memory only when there is not enough free concrete space. By waiting to make blocks concrete until we reach the casting point, we can remove the non-determinism about whether the blocks are concrete or logical. That is, blocks are always logical until the first casting point, and concrete afterward.

This also allows *ownership transfer* optimizations such as the constant propagation example in Figure IV.4, in which pointers are privately owned up until some point and then become publicly available. This optimization is performed by `clang -O2` and higher. In this example, the allocated block is initially logical and becomes concrete when cast to an integer (possibly in the call to `hash_put`). At this point, the ownership of the block is transferred from private to public. Since `a` is treated as logical up until the call to `hash_put`, we can perform constant propagation as normal before the call. (For formal details, see Section 20.5.)

On the other hand, the above model with non-deterministic allocation does not allow such optimizations for the following reasons. When the allocated block in the target is concrete, the corresponding allocation in the source must be concrete; otherwise, when the function `hash_put` casts the address of the block to an integer the source program raises out-of-memory-type behavior, while the target succeeds. Thus, you lose the ownership over the block and cannot justify the constant propagation due to the presence of `bar()`.

18.5 Operations on Pointers

In Section 18.3, we explained that the fewer blocks we make concrete, the more we can take advantage of the ownership guarantees provided by logical blocks. We have seen that operations that require pointers to have integer values force a lower bound on the number of blocks that must be made concrete. As such, we can improve our model by reducing the number of operations that require concrete addresses. We take our cue from CompCert's memory model, in which several arithmetic operations, such as

```
                              t = a + b;
d1 = a + (b - c1);      →     d1 = t - c1;
d2 = a + (b - c2);            d2 = t - c2;
```

Figure IV.5 Arithmetic optimization example II

integer-pointer addition and subtraction of pointers from pointers in the same block, are well-defined even in the absence of a concrete address (see Section 2.2).

One disadvantage of CompCert's approach to arithmetic operations is that it invalidates some important arithmetic optimizations, such as the optimization presented in Figure IV.5, by introducing logical addresses as possible values for integer-typed variables. To see this, suppose that the integer variables `a`, `b`, `c1`, and `c2` contain the same logical address $(l, 0)$. The source program shown will successfully assign $(l, 0)$ to the variables `d1` and `d2`, because `b-c1` and `b-c2` evaluate to 0. However, in the target, the variable `t` gets an unspecified value, because the addition of two logical addresses is undefined. Thus, the target has more behaviors than the source, and the optimization is invalid.

We avoid this disadvantage through the use of type checking. As in the LLVM IR, we use types to ensure that integer variables contain only integer values. This allows us to justify the full range of arithmetic optimizations on integer variables (see Section 20.5 for details), while also giving semantics to the operations on logical addresses when possible. See Section 18.8 for an example of type-dependent semantics of arithmetic operations in our model.

18.6 Dead Cast Elimination

As a result of our design decisions thus far, casts have become important effectful operations in our model, determining the points at which logical blocks are given concrete addresses. This leads to a potential problem with dead code elimination. Since casting a pointer to an integer has a side effect in memory, removing dead cast operations is not obviously justified in the hybrid model.

However, in fact, we can solve this problem and support dead cast elimination in our framework. The solution stems from our model's place in a broader compilation framework. We expect the hybrid model to be used for mid-level intermediate representations in a compiler, while the back-end low-level language will use a fully concrete model. In the hybrid model, casting a pointer to an integer has a side effect on memory, and we cannot eliminate cast operations. In the fully concrete model, however, a cast from pointer to integer is a no-op, and such casts can always be eliminated. Thus, we

```
void foo(int *p, int n) {             void foo(int *p, int n) {
  auto q = malloc(n);                   auto q = malloc(n);
  auto a = (uintptr_t) p;       →       auto a = (uintptr_t) p;
  auto r = a * 123;                     auto r = a * 123;
}                                     }
...                                   ...
foo(p, n);                      →
bar();                                bar();
```

Figure IV.6 Dead cast elimination example

can perform dead-cast-elimination optimizations in the backend.

However, we still have a problem when dead cast is combined with dead allocation. In the concrete model allocations of dead blocks cannot be removed, because otherwise an arbitrary access in the source may succeed but fail in the target.

Our solution to this problem is to remove dead casts combined with dead blocks during the translation from the hybrid to the fully concrete model. For instance, consider the dead call elimination optimization presented in Figure IV.6. This optimization is not valid when both the source and the target use the hybrid model, due to the cast operation in the function. It is also not valid when both the source and the target use the concrete memory, due to the allocation in the function. However, the optimization is valid when the source uses the hybrid model and the target uses the fully concrete model (see Section 20.5 for formal details).

Although our solution does not justify the removal of all dead casts, it should cover most of them in practice (see Section 18.7).

18.7 Drawbacks of the Hybrid Model

Although our hybrid model is designed to allow as many optimizations as possible, it still disallows some reasonable optimizations. In particular, if a function newly allocates a block and casts its address to an integer, then it loses the ownership guarantees on the block. Even if the block is still effectively locally owned, once its address is cast to an integer, we can no longer perform optimizations that rely on its locality, such as dead code elimination or constant propagation.

However, we think that blocks whose addresses are cast to integers in actual programs are unlikely to be completely local. There are few reasons to cast a local pointer to an integer unless the address will be shared with other code sections.

For instance, consider the following example of a simple program in which we might want to perform a locality optimization even after casting a pointer to an integer. The program is a variant of the example in Section 18.6, in which we cast q into an integer instead of p, so that in our model the local block becomes concrete and cannot be eliminated. Although this optimization cannot be performed in our framework, the function foo is nothing but an unpredictable number generator, and is unlikely to occur in real programs.

```
void foo(ptr p, int n) {         void foo(ptr p, int n) {
  auto q = malloc(n);              auto q = malloc(n);
  auto a = (uintptr_t) q;     →    auto a = (uintptr_t) q;
  auto r = a * 123;                auto r = a * 123;
}                                }
...                              ...
foo(p, n);                    →
bar();                           bar();
```

Another, more reasonable limitation of our model occurs when one privately uses a local block for some time, then casts its address to an integer and releases it to the public (*e.g.*, by using the integer as a key for hash table). Consider the following example, which is a variant of the example in Section 18.4, where we cast the address of a newly allocated block into an integer and use the integer as a key for hash table:

```
...                           ...
p = (int *) malloc(4);        p = (int *) malloc(4);
*p = 123;                     *p = 123;
b = (uintptr_t) p;            b = (uintptr_t) p;
bar();                   →    bar();
a = *p;                       a = *p;
hash_put(h, b, a);            hash_put(h, b, 123);
...                           ...
```

This constant propagation optimization is invalid in the hybrid model because the newly allocated block is cast to an integer before the call to bar. (It becomes valid if the cast is moved after the call to bar, though.) However, while ownership transfer optimizations of this sort are indeed performed by clang -O2, they are not performed by gcc -O2 or higher, and can be viewed as minor optimizations that are not often used.

18.8 Language Semantics

This section describes how to use the ideas of the hybrid memory model to give semantics to C-like languages, including CompCert's RTL language presented in Section 2.3.

NULL pointer We represent the NULL pointer as the logical address $(0, 0)$ and initialize the block 0 as follows:

$$m(0) = (v, p, n, c) \text{ with } v = \texttt{true}, p = 0, n = 1.$$

The only special treatment of the block 0 is that we (1) raise undefined behavior when accessing it via a load or a store; and (2) do nothing when freeing it (because `free(0)` is allowed in C).

Casting between Integers and Pointers We first define casting between integers and logical addresses via *reification* and *validity checking*. The reification function $\downarrow_m$ under memory m converts a logical address to a corresponding integer if its block in memory has a concrete address. The validity predicate valid_m checks if a logical address is inside the range of a valid block.

$$(l, i)\downarrow_m \overset{\text{def}}{=} p + i \quad \text{if } m(l) = (v, p, n, c) \land p \text{ is defined}$$

$$\text{valid}_m(l, i) \text{ iff } m(l) = (v, p, n, c) \land v = \text{true} \land (0 \le i < n)$$

Casting a logical address (l, i) into an integer first concretizes the block l and then reifies the address (l, i) if it is valid; otherwise, raises undefined behavior. Casting an integer i yields a valid logical address (l, j) that is reified to i if there is such an address; otherwise, raises undefined behavior. Note that an integer is cast to a unique address if it succeeds.

$$\begin{array}{lll} \texttt{cast2int}_m(l, i) \overset{\text{def}}{=} (l, i)\downarrow_m & \text{if } \text{valid}_m(l, i) & \{\text{after concretizing } l\} \\ \texttt{cast2ptr}_m(i) \overset{\text{def}}{=} (l, j) & \text{if } \text{valid}_m(l, j) \land (l, j)\downarrow_m = i & \end{array}$$

Computing with Logical Values We now give semantics to the binary operations based on the static types of their operands. When both operands are of type `int`, we perform ordinary integer addition, subtraction, etc. When one or more arguments are of type `ptr`, we give the operations special semantics for the well-defined cases and raise undefined behavior otherwise:

$$\begin{array}{ll} (\texttt{p + a}, m) \Downarrow (l, i_1 + i_2) & \text{if } \texttt{p} = (l, i_1) \land \texttt{a} = i_2 \\ (\texttt{a + p}, m) \Downarrow (l, i_1 + i_2) & \text{if } \texttt{a} = i_1 \land \texttt{p} = (l, i_2) \\ (\texttt{p - a}, m) \Downarrow (l, i_1 - i_2) & \text{if } \texttt{p} = (l, i_1) \land \texttt{a} = i_2 \\ (\texttt{p}_1 \texttt{ - } \texttt{p}_2, m) \Downarrow i_1 - i_2 & \text{if } \texttt{p}_1 = (l, i_1) \land \texttt{p}_2 = (l, i_2) \\ (\texttt{p}_1 = \texttt{p}_2, m) \Downarrow i_1 = i_2 & \text{if } \texttt{p}_1 = (l, i_1) \land \texttt{p}_2 = (l, i_2) \\ (\texttt{p}_1 = \texttt{p}_2, m) \Downarrow \texttt{false} & \text{if } \texttt{p}_1 = (l_1, i_1) \land \texttt{p}_2 = (l_2, i_2) \land l_1 \ne l_2 \\ & \quad \land \text{valid}_m(l_1, i_1) \land \text{valid}_m(l_2, i_2) \end{array}$$

This definition of equality is a refinement of the pointer equality given in the ISO C18 standard [37, §6.5.9p6]; for instance, it allows us to conclude that p = p even when p is not a pointer to an allocated block, while in the C standard the result of this comparison is undefined.

Hybrid and Concrete Semantics Using these definitions, we can give the usual operational semantic definitions to our language constructs, and perform memory operations (loads, stores, allocations, and casts) in the hybrid memory model. Static type checking allows us to split variables into pointer-typed variables (whose values are always logical addresses and treated as described above) and integer-typed variables (whose values are always ordinary integers and require no special handling).

We also give the language a purely concrete semantics and use it as a low-level intermediate language with the concrete memory. In this semantics, all memory blocks are concretized and all values are just integers, interpreted as either integer values or physical addresses of memory cells.

19 Fix to the LLVM and GCC miscompilation Bugs

According to our model, the LLVM bug presented in Figure I.1 is due to the first and the fourth optimizations:[2]

1. The compiler should not replace the integer comparison `pi != yi` at line `4` with the pointer comparison `&x != y+1`, because comparing integers is always safe but comparing pointers with different origins invokes undefined behavior.

4. The compiler should not replace `(int*)pi` at line `13` with `y+1`, because `y+1` is out of range.

On the other hand, the GCC bug presented in Figure IV.1 is due to the too aggressive alias analysis:

2. The compiler should not conclude that `p` is an invalid pointer. In the presence of casts between pointers and integers, casting a pure integer to a pointer may results in a valid pointer to a concrete block. Actually, `p` is alised with `&x` and thus the constant propagation optimization should not kick in.

[2] Our proposal coincides with the consensus in the bug tracker: `https://bugs.llvm.org/show_bug.cgi?id=34548`.

20 Compiler Verification Techniques and Examples

In this section, we present a compiler verification technique for proving semantic preservation under our semantics, and apply the technique to verify a number of standard optimizations.

20.1 Specification of Out of Memory

Before actually proving semantic preservation, we first generalize the notion of behaviors and observations to make compiler verification more faithful in the presence of out of memory. Recall that out of memory may be raised by pointer-to-integer cast, as we discussed in Section 18.4. It is worth noting that out of memory does not occur in CompCert because it uses a purely logical memory model that has infinite memory.

First, we regard out of memory as *the empty set of behaviors* (*i.e.*, no behaviors), following CompCertTSO [74]. This is because we should allow the target program to run out of memory even if the source program does not, because register allocation—one of the most important compiler optimizations—may increase the program's memory requirements. Furthermore, since running out of memory is not that serious a problem for programmers as is genuine undefined behavior (*e.g.*, accessing freed memory), the target behavior of a source program running out of memory should not be arbitrary.

Second, we allow programmers to observe *partial behaviors*, possibly before discarding behavior due to running out of memory, in addition to (whole) behaviors presented in Section 2.1:

- A partial execution of the program that has produced a finite sequence of I/O events, $e_1, \cdots, e_n, \texttt{partial}$.

Partial behaviors allow programmers to observe I/O events prior to an out-of-memory error, which is more faithful to the real world because it is absurd to discard I/O events before running out of memory. As before, refinement is defined as inclusion of the set of (possibly partial) behaviors of the target program into that of the source program. As a consequence, compilers should guarantee that the target program always performs a prefix of the events the source program could have performed.

We have to admit that our treatment of out of memory—as well as that of all the other existing formal semantics—is not entirely satisfactory. Firstly, our semantics may not justify those optimizations that reduce (concrete) memory consumption instead of increasing it, such as dead cast elimination.[3] Furthermore, the ISO C18 standard says

[3] Notice that register promotion and dead allocation elimination does not reduce concrete memory consumption in our hybrid model because they remove purely logical memory blocks.

that `malloc` returns zero in case of out of memory [37, §7.22.3p1], instead of allowing no behaviors.[4] We leave defining a more proper semantics of out of memory as a future work.

20.2 Running Example & Informal Verification

Now we actually prove semantic preservation for the following example transformation, which is indeed performed by `clang -O2`. This transformation involves four different optimizations: constant propagation (CP), dead load elimination (DLE), dead store elimination (DSE), and dead allocation elimination (DAE).

```
    foo(int *p) {                              foo(int *p) {
1:    auto q = (int *) malloc(4);                          // DAE
2:    *q = 123;                                            // DSE
3:    bar(p);                             →      bar(p);
4:    auto a = *q;                                         // DLE
5:    *p = a;                                    *p = 123; // CP
    }                                          }
```

We will argue that at each line in the two versions of `foo` (source and target), the effects of the instruction executed (if any) are equivalent. To do so, we will assume an initial relationship between the memory of the source and target programs, and show that some variant of that relationship persists throughout the function, relying on any call to other functions to maintain a similar relationship.

This relationship will designate one section of each memory as *public*, and require that related locations in the source and target public memories have equivalent values; it will also designate a *private* section of each memory, such that the source program can make changes to its private memory when the target does not make corresponding changes and vice versa. For the technical details of this relation and our notion of equivalence, see Section 20.3.

We begin at line 1 by assuming the following conditions on the memory of the source and target programs (see Figure IV.7 (a)):

assume (equivalent arguments) the parameter `p` contains *equivalent* arguments v_{src} in the source and v_{tgt} in the target;

[4] The ISO C18 standard semantics, however, is also unsatisfactory in that it does not justify dead allocation eliminations, *e.g.*, if a dead malloc is eliminated, then the source may have more allocated blocks than the target has and thus a subsequent allocation may return zero in the source but return a valid pointer in the target.

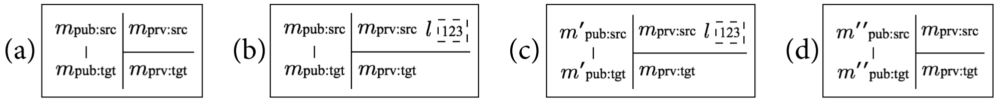


Figure IV.7 Memory invariants for the running example

assume (equivalent public memories) there are a set of memory blocks $m_{\text{pub:src}}$ in the source and $m_{\text{pub:tgt}}$ in the target that are *equivalent* and publicly accessible by arbitrary functions;

assume (source private memory) there is a disjoint set of blocks $m_{\text{prv:src}}$ in the source, each of which is exclusively owned by a single function;

assume (target private memory) there is a disjoint set of blocks $m_{\text{prv:tgt}}$ in the target, each of which is exclusively owned by a single function.

After executing line 1, we add the newly allocated block (call it l) to the private source memory $m_{\text{prv:src}}$. It is important to note that we can add the block l to the private source memory because it is a fresh logical block and thus exclusively owned by `foo`. After executing line 2, the block l contains 123 (see Figure IV.7 (b)).

At line 3, we guarantee that the function calls to `bar` are *equivalent* as follows:

guarantee (equivalent arguments) the arguments v_{src} and v_{tgt} to `bar` are equivalent;

guarantee (equivalent public memories) $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$, which are equivalent and publicly accessible;

guarantee (source private memory) each location in $m_{\text{prv:src}} \uplus [l \mapsto 123]$, is exclusively owned by a single function;

guarantee (target private memory) each location in $m_{\text{prv:tgt}}$ is exclusively owned by a single function.

When the calls to `bar` return, we can assume that the new public memories are equivalent to each other (though they may not be the same as the previous public memories), and the private memories are untouched (see Figure IV.7 (c)):

assume (equivalent public memories) we have new public memories $m'_{\text{pub:src}}$ and $m'_{\text{pub:tgt}}$, which are *evolved* from $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$ (see Section 20.4 for the definition of memory evolution), and are equivalent and publicly accessible;

assume (source private memory) $m_{\text{prv:src}} \uplus [l \mapsto 123]$ is unchanged;

assume (target private memory) $m_{\text{prv:tgt}}$ is unchanged.

At line 4, we load the value 123 from the source's private memory and store it in the variable `a`. At line 5, in the source, we store the value of `a`, which is 123, in the memory cell located at the address $v_{\texttt{src}}$. In the target, we store the constant 123 in the cell at $v_{\texttt{tgt}}$. Since we stored equivalent values at equivalent locations $v_{\texttt{src}}$ and $v_{\texttt{tgt}}$, we will have equivalent public memories $m''_{\text{pub:src}}$ and $m''_{\text{pub:tgt}}$, while leaving the private memories unchanged.

Finally, we return to the callers of `foo` with (see Figure IV.7 (d))

guarantee (equivalent public memories) $m''_{\text{pub:src}}$ and $m''_{\text{pub:tgt}}$ that are evolved from $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$, and are equivalent and publicly accessible;

guarantee (source private memory) $m_{\text{prv:src}}$;

guarantee (target private memory) $m_{\text{prv:tgt}}$,

where we can ignore the block l because it is not going to be used any more. Note that we here guarantee foo returns with the same private memories it was given initially, as we assumed the same property for the function `bar` after line 3.

20.3 Memory Invariants

As informally discussed above, we prove behavior refinement using a memory invariant that places conditions on public memories (which must be equivalent in the source and target programs) and private memories (which can differ between them). We now formally define the notion of memory equivalence used in our informal example, and the conditions on the private memories.

If you are familiar with CompCert's verification, you may regard the memory invariant is a generalization of CompCert's *memory injection* [57] to account for private memories, function-modular reasoning, and finite memory.[5] Furthermore, our conditions for concrete blocks are inspired from CompCertTSO's support for finite memory [74]. See Section 21 for more comparisons.

[5] In fact, the memory invariant is also a simplification of CompCert's memory injection in the treatment of public memories: the public memories in the source and the target should coincide in the invariant, while they may differ in CompCert's memory injection. Though it is straightforward to generalize the invariant in this dimension.

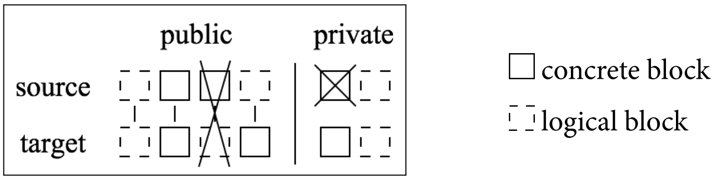


Figure IV.8 Memory invariants for hybrid model

Memory Equivalence We define a more relaxed notion of equivalence than simple equality, which would be too strong in the presence of (unrelated) private memories. We say that a set of blocks m_{src} in the source is equivalent to a set of blocks m_{tgt} in the target when they satisfy the following conditions. First, there should be a bijection, say α, between the block identifiers in m_{src} and those in m_{tgt}. Second, corresponding blocks (*i.e.*, those related by α) should have the same size and validity, and the values they hold at each offset should be *equivalent*. Values are equivalent (w.r.t. α) when either both are the same integer, or they are logical addresses that are at the same offset in corresponding blocks (w.r.t. α). When m_{src} and m_{tgt} are equivalent in this sense, we write $m_{\mathrm{src}} \simeq_\alpha m_{\mathrm{tgt}}$.

The condition on the concrete addresses of corresponding blocks merits further explanation. We have four possible cases regarding whether two corresponding blocks are concrete or logical (see the *public* side of Figure IV.8). The first case in the figure (*i.e.*, source: logical, target: logical) obviously should be allowed. The second case (*i.e.*, source: concrete, target: concrete) should also be allowed but only when the concrete addresses coincide.

The third case (*i.e.*, source: concrete, target: logical) should not be allowed. To allow this case would be to allow the source memory to contain more concrete blocks than the target, which leads to two problems: (1) an arbitrary concrete memory access may succeed in the source but fail in the target; and (2) a pointer-to-integer cast may raise out-of-memory in the source but succeed in the target. In both cases, the target may have more behaviors than the source, which is disallowed. On the other hand, the final case (*i.e.*, source: logical, target: concrete) is allowed because the situation is exactly the opposite: the source may have more behaviors than the target, which is allowed.

Private Memory For blocks in private memories, we have four possible cases regarding whether the block is in the source or the target, and whether the block is concrete or logical (see *private* side of Figure IV.8). All the cases are allowed except for source private memory blocks that are concrete, for the same reason that blocks that are concrete in the source and logical in the target are not allowed in memory equivalence: the source memory should not contain more concrete blocks than the target memory.

Memory Invariants A memory invariant β consists of (1) a bijection α between their block identifiers, (2) the source's private memory $m_{\mathrm{prv:src}}$, and (3) the target's private memory $m_{\mathrm{prv:tgt}}$. An invariant $\beta = (\alpha, m_{\mathrm{prv:src}}, m_{\mathrm{prv:tgt}})$ holds on a pair of memories m_{src} and m_{tgt} when they contain the private sections $m_{\mathrm{prv:src}}$ and $m_{\mathrm{prv:tgt}}$ and some public sections $m_{\mathrm{pub:src}}$ and $m_{\mathrm{pub:tgt}}$ such that:

$$\begin{array}{l}(m_{\mathrm{src}} \supseteq m_{\mathrm{pub:src}} \uplus m_{\mathrm{prv:src}}) \ \wedge \ (m_{\mathrm{tgt}} \supseteq m_{\mathrm{pub:tgt}} \uplus m_{\mathrm{prv:tgt}}) \wedge \\ m_{\mathrm{pub:src}} \simeq_{\alpha} m_{\mathrm{pub:tgt}}\end{array}$$

where $\uplus$ and $\subseteq$ are the disjoint union and the subset relation.

20.4 Proving Simulation

We are now ready to present our reasoning principle formally. Our basic approach is to verify programs via local simulation in the style of [33].

A function, say `foo`, in the source and target is locally simulated if it satisfies the following conditions. First, consider a typical lifecycle of the source and target functions:

$$\begin{array}{llll}
\texttt{foo(..) \{} & & \texttt{foo(..) \{} & /\!/ \boxed{\beta_{\mathrm{s}}} \\
\quad \texttt{...} & & \quad \texttt{...} & /\!/ \ \beta_{\mathrm{c}} \quad \beta_{\mathrm{s}} \sqsubseteq \beta_{\mathrm{c}} \\
\quad \texttt{bar(..);} & & \quad \texttt{bar(..);} & /\!/ \boxed{\beta_{\mathrm{r}} \quad \beta_{\mathrm{c}} \sqsubseteq \beta_{\mathrm{r}} \ \wedge \ \beta_{\mathrm{c}} =_{\mathrm{prv}} \beta_{\mathrm{r}}} \\
\quad \texttt{...} & \rightarrow & \quad \texttt{...} & /\!/ \ \beta'_{\mathrm{c}} \quad \beta_{\mathrm{r}} \sqsubseteq \beta'_{\mathrm{c}} \\
\quad \texttt{gee(..);} & & \quad \texttt{gee(..);} & /\!/ \boxed{\beta'_{\mathrm{r}} \quad \beta'_{\mathrm{c}} \sqsubseteq \beta'_{\mathrm{r}} \ \wedge \ \beta'_{\mathrm{c}} =_{\mathrm{prv}} \beta'_{\mathrm{r}}} \\
\quad \texttt{...} & & \quad \texttt{...} & /\!/ \ \beta_{\mathrm{e}} \quad \beta'_{\mathrm{r}} \sqsubseteq \beta_{\mathrm{e}} \ \wedge \ \beta_{\mathrm{s}} =_{\mathrm{prv}} \beta_{\mathrm{e}} \\
\texttt{\}} & & \texttt{\}} &
\end{array}$$

Here boxed conditions are assumed and the others are guaranteed.

First, in `foo`, unknown functional calls such as `bar(..)` and `gee(..)` should be synchronized (*i.e.*, when the target calls `bar`, the source should call `bar` as well). Note that when a known function is called, the verifier can either step into the called function and reason about its code, or treat it as an unknown function call.

Next, at the entry point of `foo`, we assume that we are given memories satisfying a given invariant β_{s}, and equivalent arguments w.r.t. the bijection in β_{s}. Then, we execute the code of foo in the source and the target until the first unknown function call to `bar(..)`. Here we have to show that there is some invariant β_{c} that holds on the current memories and that the arguments to the function `bar` are equivalent w.r.t. the bijection in β_{c}.

Here, we also have to show that the current memories are *evolved* from the memories given initially by showing that the current invariant β_c is a *future invariant* of the initial one β_s (denoted $\beta_s \sqsubseteq \beta_c$). We say that β_c is a future invariant of β_s when satisfying the following conditions, which rule out changes to the memory that cannot be caused by the language's operational semantics. First, the bijection in β_c should include the bijection of β_s because logical blocks cannot be removed during execution (a block becomes invalid rather than removed when it is freed). Second, the other conditions on the public memories in β_s and β_c are that (1) the size of a block does not change between β_s and β_c, (2) an invalid block in β_s cannot become valid in β_c, and (3) a concrete block in β_s cannot become logical in β_c. However, it is important to note that the contents of public memories can change between β_s and β_c because the operational semantics allows to update values in memory.

Then, we consider the case when the unknown function successfully returns. We can assume that the memories at return time also satisfy some *future* invariant β_r. We can also assume that the function `bar` does not change the private memories in β_c (denoted $\beta_c =_{\mathrm{prv}} \beta_r$) because there is no way for `bar` to access them in our hybrid model.

We continue through the function, evolving our invariant at non-call steps and performing similar reasoning at other call sites such as `gee(..)`. Finally, when `foo` returns to its caller, we have to show that there is some *future* invariant β_e that holds on the current memories. Furthermore, we have to show that we did not change the private memories given in the initial invariant β_s (*i.e.*, $\beta_s =_{\mathrm{prv}} \beta_e$). This condition is necessary because, as seen above, we assume that this property holds at the end of any other function call. In this way, we construct a local simulation proof for the `foo`.

20.5 Examples

In this section, we show how to verify the examples shown in Section 18. All results here are fully formalized in Coq.

Arithmetic Optimization I Consider the transformation in Figure IV.2. If we assume that integer variables only contain integer values, not logical addresses, the instruction `a = (a - b) + (2 * b - b)` has no effect on the value of `a` and is equivalent to no operation, so the optimization is trivially correct.

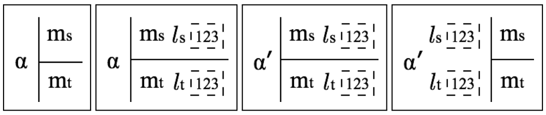
How do we know that integer variables only contain integer values? The straightforward answer is that our language is statically type-checked, as in the LLVM IR. However, the key reason why this is possible is that in the hybrid model we actually turn logical addresses into integers when they are cast to `int`, rather than placing logical addresses in integer variables. Also, when we load a value from memory to an integer variable (resp. a pointer variable), if the loaded value is a logical address (resp. an integer value), we raise undefined behavior (*i.e.*, *error*). In other words, the hybrid model induces a form of dynamic type checking in languages that use it. This allows us to verify integer arithmetic optimizations as in this example.

Dead Code Elimination Consider the transformation in Figure IV.3. This example is similar to the previous one. Since we can assume that integer-typed variables contain only integers, the execution of the call `foo(a)` does not have any side effects. Furthermore, because we know the code of the function `foo`, we do not need to treat it as an unknown function call. Rather, we just step into the code of the function `foo` and execute it in the source.

Ownership Transfer Consider the transformation in Figure IV.4. This example is similar to the running example in Section 20.2.

Assume that the first invariant below holds before the `malloc`. After allocating blocks l_s in the source and l_t in the target, and storing 123 in both blocks, we can move the blocks l_s and l_t into the private sections of the invariant because they are logical and disjoint from the public sections, yielding the second invariant below. Next we call the function `bar`. When it returns, we can assume that the third invariant holds (*i.e.*, the private sections are untouched). After loading, the variable `a` will contain 123, since `p` contains the logical address $(l_s, 0)$ in the source and $(l_t, 0)$ in the target.

Next, when we call `hash_put`, we have to make sure that the arguments are equivalent. The first arguments are equivalent because we assume that we start with equivalent values in variables, and the third arguments are equivalent because `a` contains 123. To show that the second arguments, $(l_s, 0)$ and $(l_t, 0)$, are equivalent, we move the blocks from the private sections to the public section and extend the bijection α' to relate l_s and l_t) (the fourth invariant below). Such ownership transfer from the private sections to the public section is allowed because the future invariant relation ($\sqsubseteq$) requires only the bijection to be non-decreasing, not the private sections.

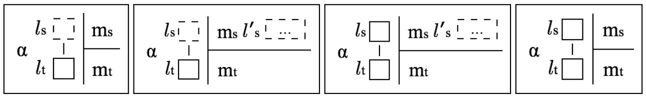


Arithmetic Optimization II We can easily verify the transformation in Figure IV.5 for the same reason as in Figure IV.2: because we can assume that all integer variables contain integer values.

Dead Cast Elimination Consider the transformation in Figure IV.6, in the case in which the source uses the hybrid model and the target uses the concrete model.

We begin by assuming that the first invariant below holds before the call to `foo`, where the variable `p` contains equivalent addresses (l_s, i) in the source and $(l_t, i)\downarrow_m$ in the target. Note that the block l_t is concrete, since the target is using the fully concrete model. After the allocation of a block, say l'_s, in `foo` in the source, we move it to the source's private memory, yielding the second invariant below. Here it is important to note that if the source was using the concrete model, we could not move the block l'_s into the private section because l'_s would be concrete, which would invalidate our proof.

After the cast, the block l_s becomes concrete, yielding the third invariant below. Here it is important to note that if the target language was using the hybrid model and l_t were logical, then we would produce an invariant in which l_s is concrete and l_t is logical, which would be an invalid invariant. After `foo` returns, we simply drop the block l'_s from the source private section because we do not use it, yielding the fourth invariant below. Then we can proceed to verify the rest of the code.



Identity Compilers As a sanity check for our reasoning principles, we wrote an identity compiler from our language with the hybrid model to itself, and a simple compiler from our language with the hybrid model to the same language with the concrete model. The latter compiler just eliminates dead casts of the form `_ = (int) p`. We successfully verified these two compilers in Coq using our reasoning principles.

21 Discussion

21.1 Implementation and Experiment Details

Coq Formalization All the proofs reported in this paper have been fully formalized in Coq and can be found in the project webpage. Our Coq formalization is about 10,000 lines of code, excluding empty lines and library code. The formalization took about 2 person-months to complete.

Optimization Examples All optimization examples presented in the paper are performed by Clang 3.4.2 and/or GCC 4.8.3. Examples in C and their compilation results can be found in the project webpage.

21.2 Related Work

Formal Memory Models There have been numerous efforts to formalize C semantics, both from the perspective of clarifying the specification and defining implementations with formal semantics [63, 56, 26, 48, 29]. These invariably use variations of the logical memory model, where each allocation is associated with some abstract identifier and pointers consist of an identifier and some path representing an offset into the memory block, except for the work of Norrish [63] which uses the concrete model.

Comparison with CompCert CompCert [56, 57] and its various extensions currently allow casting pointers to and from integers, but the semantics preserves the logical representation of pointers after the cast. As a result, integer variables can contain not only normal 32-bit integers, but also logical pointer representations. In the higher-level languages (CompCert C and Clight), performing arithmetic on cast pointer is treated as a program error, whereas in the low-level languages (from Cminor down to assembly), adding and subtracting integer values from converted pointers is defined and affects only the offset into the pointer's logical block. There has also been work on extending the semantics to support pointer fragments to allow, for example, `memcpy` to work on memory containing pointers [47], but these extensions still cannot fully support arithmetic operations on pointer values that have been cast to an integer type.

Comparison with CompCertTSO The CompCertTSO compiler [74] extends CompCert's Clight language with threading and atomic memory primitives following the x86-TSO relaxed memory model. Similar to us, CompCertTSO's memory model also supports finite memory, but uses a different mechanism to do so. It has a distinguished logical block, where the offset serves as essentially a concrete memory address. During compilation, all memory operations are lowered to act only on a single finite logical block. This allows the source and target languages, with infinite and finite memory respectively, to share a single memory model, and simplifies the correctness statements by removing the need for CompCert's memory injections. CompCertTSO handles pointer-integer casts in the same way as CompCert, with the same limitations.

Comparison with the Symbolic Value Approach Most recently, Besson et al. have proposed an extension to CompCert's memory model that gives semantics to bit-masking operations on pointers and uninitialized values [18]. Their approach involves adding lazily-evaluated symbolic expressions, including arbitrary operations on the representation of pointers, to the class of semantic values. Symbolic values are forced whenever a concrete value is needed to take a step, for example to access memory through a pointer or in the guard of a conditional. The mapping is performed by a normalization function given as a parameter of the semantics. The normalization function is partial, and is only defined precisely when the symbolic value evaluates to a unique result under every assignment from logical block identifiers to concrete addresses (subject to some validity conditions).

The semantics of Besson et al. is necessarily deterministic: non-determinism is interpreted as undefined behavior, while our model captures the non-deterministic allocation of concrete addresses. Furthermore, their semantics is complex and indeed intractable: their normalization is implemented with an SMT solver, and the semantics in general is too complex to serve as a mental model for ordinary C programmers. Normalization in our semantics, on the other hand, is a straightforward translation from pointers to concrete blocks and integers.

Most importantly, while their approach gives semantics to non-strictly-conforming C programs involving bit-masking of pointers and uninitialized values, it fails to define useful programs that use integer-pointer casts. Consider the `hash_put` example discussed in Section 18.4, where a pointer is hashed and then presumably used to index into an array. Since the resulting memory location will depend on the concrete layout of memory, the resulting program will have undefined behavior in their semantics. In general, any program that displays non-determinism due to the concretization of pointers in our model is necessarily undefined in Besson et al.'s model.

21.3 Compatibility

Compatibility with Other C Language Features There are numerous other C language features that have some interaction with the memory model. Some of them, such as *indeterminate values* [37, §3.19.2p1], *dangling pointers* [37, §6.2.4p2], and *infinite loops with no side-effects* [37, §6.8.5p6], have semantics that are largely orthogonal to the pointer concretization used in our hybrid model. Similarly, our model explicitly allows *unsafely-derived pointers*, which are permitted in C18 and implementation-defined in C++17 [36, §6.6.4.4.3]. We allow them in order to support low-level programming idioms such as XOR linked lists and compressed oops in HotSpot JVM.

Our paper does not directly address threads, so we cannot claim with certainty that the model extends to handle them. However, we see no obstacles in this direction, and the hybrid model is similar to CompCertTSO, which does support a weak memory model and threads, so we are optimistic that this extension to the semantics should follow similarly.

A few language features require some adaptation of our memory model. For instance, we can adapt the hybrid model to support *union types* and *strict aliasing*, following Krebbers' technique [45], which works regardless of whether the model is concrete, logical or hybrid.

As another example, in C, `char*` is a "universal" pointer type, which allows efficient bulk data moves via `memcpy`. Krebber's variant of CompCert [47] already supports this semantics using a logical memory, and the hybrid model is compatible with that solution. Briefly: we let `char` types store byte-indexed logical values (such as $(l, 10) : 2$, which denotes the second byte of the logical address $(l, 10)$). This strategy works because a `char` is implicitly cast to an integer when used in arithmetic operations, and thus we can simply treat these casts as side-effecting (*i.e.*, concretizing the logical addresses). This approach lose (almost) no optimization opportunities because byte-indexed logical addresses are typically loaded from the memory and thus (mostly) already treated as public by the compiler.

Compatibility with Alias Analyses The hybrid model is largely compatible with common alias analyses. For instance, it can be used to justify *size-based alias analysis*, which considers pointers to differently-sized objects as distinct. For example, in the code below, there is no alias between `p` and `q`: even if `q` points to the block pointed to by `p`, loading or storing a double value in the block will fail since the block is not big enough to contain double values.

```
int *p = malloc(sizeof(int));
double *q = foo(p);     // no alias between p and q
```

It also justifies *freshness-based alias analysis*, which assumes that the result of `malloc` is distinct from all other pointers. The following example of constant propagation is valid in the hybrid model since `q` points to a fresh block that is different from the block pointed to by `p`. It is important to note that there is no alias between `p` and `q` even after the fresh block is concretized. The reason is because even if `p` and `q` may be cast to the same integer, they still point to different blocks as pointer values.

```
void foo(int *p) {                     void foo(int *p) {
  auto q = (int *) malloc(4);            auto q = (int *) malloc(4);
  auto a = (uintptr_t) q;                auto a = (uintptr_t) q;
  auto b = *p;                   →       auto b = *p;
  *q = 123;                              *q = 123;
  auto r = *p;                           auto r = b;          // CP
}                                      }
```

21.4 Impact

In the course of doing this research, we discovered the GCC bug presented in Figure IV.1. With this bug, we persuaded compiler writers that (1) it is subtle to define the semantics of casts between pointers and integers, and (2) it is safe to turn off some alias analyses that are too aggressive to peacefully coexist with other optimizations.

Our idea to give semantics to more programs involving pointer operations—namely, using concrete and logical blocks at the same time—has subsequently been refined by follow-up papers by other researchers [54, 60], which are accompanied with promising revision proposals to the LLVM compiler and ISO C standard.

We intend to use this model for compiler verification tasks, extending the range of common optimizations that can be verified. Ultimately, we would like to generalize Crellvm [43], which is a credible compilation framework for LLVM, to support casts between pointers and integers. We would also like to integrate our model with CompCert and use it to justify new CompCert optimizations. We believe that our ideas are readily applicable to CompCert(TSO) and related projects like Vellvm [83, 84] because our memory model and notion of memory invariant are technically very close to CompCert's. (Vellvm also uses CompCert's memory model.) Essentially, all that would have to change in the proofs are the cases handling pointer to integer casts.

Chapter V

Epilogue

22 Conclusion

Genesis The genesis of this dissertation traces back to 2014, when my supervisor Prof. Chung-Kil Hur ran a seminar on CompCert [55]. CompCert is exciting because it demonstrated formal semantics and compiler verification research scale up to real-world applications. One day, he asked what are the concrete addresses of global variables in CompCert C, and I answered "I don't know." So we read CompCert code together, and realized that CompCert C does not assign concrete addresses to global variables because it does not support cast between integers and pointers. Then we started to develop a formal semantics of the feature, which ended up being my first first-author paper [41].

It was exciting to advance the state-of-the-art of C formal semantics, and it naturally became my dissertation topic. I focused on low-level features because they are difficult to capture in a formal semantics due to the conflict among many "stakeholders." To understand them, I began to read C/C++ standards, Linux kernel and LLVM mailing lists, and even architecture ISA manuals which I didn't expect to do so at the beginning of my graduate school life.

Contributions As a result, we developed formal semantics of three low-level features of C crucially used in systems programming, namely relaxed-memory concurrency, separate compilation, and cast between integers and pointers. Our semantics adequately balances the conflicting desiderata of programmers, compilers, and hardware in that it (1) supports the features' common usage patterns and reasoning principles for programmers, and (2) provably validates major compiler optimizations at the same time. To establish confidence in our formal semantics, we have formalized most of our key results in Coq.

Impact Our formal semantics had concrete impacts on both academia and industry. In academia, we and others published follow-up papers that (1) refine our semantics to account for more usage patterns and compiler optimizations, (2) provide more evidences that show our semantics serves programmers, compilers, and hardware well, or (3) apply the key idea of our semantics to other languages. In industry, (1) we discovered bugs and proposed fixes to bugs in GCC, LLVM, and CompCert, (2) we provided informed opinions to the discussion on C/C++ and LLVM IR language standards, and (3) our verification technique was adopted in CompCert.

Future Work However, by Meyer's standard, where real success is "changing the way the IT industry develops software", "the story told in this article is one of glaring, unremitted and probably definitive failure" [17]. CompCert is changing the way safety-critical software is developed [14, 50]. In the similar spirit, we would like our semantics to help system programmers, thereby changing the way systems are developed in general. To this end, we would like to pursue the following directions for future work:

- *Refining Semantics*: Our semantics, as well as all the other C formal semantics, does not capture all the real-world practices of low-level features and compiler optimizations. While it is impossible to do so, it is crucial to capture at least widely used patterns and performance-critical optimizations. Otherwise, industrial developers will not approve our semantics and consider it a "toy" language. In order to do so, we need to clearly understand each low-level feature's motivation and use cases.

- *Standardizing Semantics*: Developing a semantics should take into account a lot of stakeholders, and thus discussion on the semantics can easily diverge to too many proposals. For mainstream languages like C, C++, and LLVM IR, standardization process is introduced to help the discussion to converge to a consensus. We should also propose our semantics to the standardization process, thereby exchanging informed opinions on the semantics.

- *Developing Tools*: It is unlikely that semantics alone improves the life of industrial developers, because semantics is a theoretical artifact while developers need practical tools that help them to design and implement programs, and catch and prevent bugs. Thus it is crucial to develop such tools—including model checkers, static analyzers, and program logics—based on semantics and metatheory.

Bibliography

[1] Coq development and supplementary material for this thesis. https://sf.snu.ac.kr/jeehoon.kang/thesis.

[2] GCC, the GNU compiler collection. https://gcc.gnu.org/.

[3] The LLVM compiler infrastructure project. https://llvm.org/.

[4] A promising semantics for relaxed-memory concurrency. https://github.com/nikomatsakis/rust-memory-model/issues/32.

[5] Promote memory to register. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register.

[6] Register allocation. https://en.wikipedia.org/wiki/Register_allocation.

[7] Relaxed-memory concurrency synchronization patterns. https://jeehoonkang.github.io/2017/08/23/synchronization-patterns.html.

[8] Why is this legal? https://groups.google.com/forum/?hl=en#!msg/comp.std.c/ycpVKxTZkgw/S2hHdTbv4d8J.

[9] JSR 133. Java memory model and thread specification revision, 2004. http://jcp.org/jsr/detail/133.jsp.

[10] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. 17th Annual International Symposium on Computer Architecture*, ISCA 1990, pages 2–14. ACM, 1990.

[11] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[12] A. W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[13] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 55–66. ACM, 2011.

[14] R. Bedin França, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Towards optimizing certified compilation in flight control software. In *Workshop on Predictability and Performance in Embedded Systems (PPES 2011)*, volume 18 of *OpenAccess Series in Informatics*, pages 59–68. Dagstuhl Publishing, 2011.

[15] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[16] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP*, 2014.

[17] Bertrand Meyer. Fourteen years of software engineering at eth zurich. 2016.

[18] F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for C using symbolic values. In *APLAS*, 2014.

[19] H.-J. Boehm. P1217R0: Out-of-thin-air, revisited, again. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1217r0.html`, 2018.

[20] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proc. Workshop on Memory Systems Performance and Correctness*, MSPC 2014, pages 7:1–7:6. ACM, 2014.

[21] S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *Proc. 15th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2017, 2017.

[22] The CompCert C compiler. `http://compcert.inria.fr/`.

[23] K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 623–636. ACM, 2015.

[24] D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek. Plan B: A buffered memory model for Java. In *Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2013, pages 329–342. ACM, 2013.

[25] S. Dolan, K. C. Sivaramakrishnan, and A. Madhavapeddy. Bounding data races in space and time. In *PLDI*, 2018.

[26] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, 2012.

[27] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 608–621. ACM, 2016.

[28] J. Gosling, B. Joy, and G. Steele. The Java language specification, Edition 1.0, Aug. 1996. http://titanium.cs.berkeley.edu/doc/java-langspec-1.0/.

[29] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *PLDI*, 2014.

[30] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, 2015.

[31] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *POPL*, 2011.

[32] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[33] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[34] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The transitive composability of relation transition systems. Technical Report MPI-SWS-2012-002, MPI-SWS, 2012.

[35] INRIA. The Coq proof assistant. http://coq.inria.fr/.

[36] ISO. ISO/IEC 14882:2017: Programming languages — C++. https://www.iso.org/standard/68564.html.

[37] ISO. ISO/IEC 9899:2018: Programming languages — C. https://www.iso.org/standard/74528.html.

[38] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *ESOP*, pages 307–326, 2010.

[39] A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. In *Proc. IEEE Logic in Computer Science*, LICS 2016, 2016.

[40] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL*, 2017.

[41] J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal c memory model supporting integer-pointer casts. In *PLDI*, 2015.

[42] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *POPL*, 2016.

[43] J. Kang, Y. Kim, Y. Song, J. Lee, S. Park, M. D. Shin, Y. Kim, S. Cho, J. Choi, C. Hur, and K. Yi. Crellvm: verified credible compilation for LLVM. In *PLDI*, 2018.

[44] Khronos Group. The OpenCL specification, Version 2.1, 2015.

[45] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, 2013.

[46] R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

[47] R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, 2014.

[48] R. Krebbers and F. Wiedijk. A formalization of the C99 standard in HOL, Isabelle and Coq. In *CICM*, 2011.

[49] R. Kumar, M. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL*, 2014.

[50] D. Kästner, U. Wünsche, J. Barrho, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*, 2018.

[51] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 649–662. ACM, 2016.

[52] O. Lahav and V. Vafeiadis. Explaining relaxed memory models with program transformations. In *Proc. 21st International Symposium on Formal Methods*, FM 2016, 2016.

[53] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in c/c++11. PLDI 2017, 2017.

[54] J. Lee, C. Hur, R. Jung, Z. Liu, J. Regehr, and N. P. Lopes. Reconciling high-level optimizations and low-level code in LLVM. *PACMPL*, 2(OOPSLA), 2018.

[55] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[56] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[57] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.

[58] A. Lochbihler. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12:1–12:65, 2014.

[59] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proc. 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2005, pages 378–391. ACM, 2005.

[60] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring c semantics and pointer provenance. *PACMPL*, 3(POPL), 2019.

[61] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *ICFP*, 2015.

[62] B. Norris and B. Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proc. 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pages 131–150. ACM, 2013.

[63] M. Norrish. C formalised in HOL. Computer Laboratory Technical Report 453, University of Cambridge, Nov. 1998.

[64] P. Ou and B. Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *PACMPL*, 2(OOPSLA), 2018.

[65] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs 2009, pages 391–407. Springer, 2009.

[66] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP*, 2014.

[67] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 622–633. ACM, 2016.

[68] A. Podkopaev, O. Lahav, and V. Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. In *POPL*.

[69] A. Podkopaev, I. Sergey, and A. Nanevski. Operational aspects of C/C++ concurrency. *CoRR*, abs/1606.01400, 2016.

[70] C. Pulte, J. Pichon-Pharabod, J. Kang, S.-H. Lee, and C.-K. Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *PLDI*, 2019 (conditionally accepted).

[71] T. Ramananandro, Z. Shao, S. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. In *CPP*, 2015.

[72] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *Proc. 22nd European Conference on Object-Oriented Programming, ECOOP 2008*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.

[73] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[74] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013.

[75] J. Souyris. Industrial use of CompCert on a safety-critical software product, Feb. 2014. Talk slides available at: http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf.

[76] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *POPL*, 2015.

[77] K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis. A separation logic for a promising semantics. In *ESOP*, 2018.

[78] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 209–220. ACM, 2015.

[79] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *OOPSLA*, 2014.

[80] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, 2013.

[81] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[82] Y. Zhang and X. Feng. An operational happens-before memory model. *Frontiers of Computer Science*, 10(1):54–81, 2016.

[83] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.

[84] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI*, 2013.

초록

주류 C 컴파일러들은 프로그램의 성능을 높이기 위해 공격적인 최적화를 수행하는데, 그런 최적화는 저수준 기능을 사용하는 프로그램의 행동을 바꾸기도 한다. 불행히도 C 언어를 디자인할 때 저수준 기능과 컴파일러 최적화를 적절하게 조화시키기가 굉장히 어렵다는 것이 학계와 업계의 중론이다. 저수준 기능을 위해서는, 그러한 기능이 시스템 프로그래밍에 사용되는 패턴을 잘 지원해야 한다. 컴파일러 최적화를 위해서는, 주류 컴파일러가 수행하는 복잡하고도 효과적인 최적화를 잘 지원해야 한다. 그러나 저수준 기능과 컴파일러 최적화를 동시에 잘 지원하는 실행의미는 오늘날까지 제안된 바가 없다.

본 박사학위 논문은 시스템 프로그래밍에서 요긴하게 사용되는 저수준 기능과 주요한 컴파일러 최적화를 조화시킨다. 구체적으로, 우린 다음 성질을 만족하는 느슨한 동시성, 분할 컴파일, 정수-포인터 변환의 실행의미를 처음으로 제안한다. 첫째, 기능이 시스템 프로그래밍에서 사용되는 패턴과, 그러한 패턴을 논증할 수 있는 기법을 지원한다. 둘째, 주요한 컴파일러 최적화들을 지원한다. 우리가 제안한 실행의미에 자신감을 얻기 위해 우리는 논문의 주요 결과를 대부분 Coq 증명기 위에서 증명하고, 그 증명을 기계적이고 엄밀하게 확인했다.

주요어: C, 실행의미, 컴파일러 검증, 느슨한 동시성 메모리, 분할 컴파일, 정수-포인터 변환
학번: 2013-20737