TU WIEN Informatics

Supporting Register Pairs in CompCert

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alexander Loitzl, BSc
Matrikelnummer 11805113

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Math. Dr.techn Florian Zuleger

Wien, 8. Mai 2024

Alexander Loitzl	Florian Zuleger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Supporting Register Pairs in CompCert

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Alexander Loitzl, BSc
Registration Number 11805113

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Math. Dr.techn Florian Zuleger

Vienna, May 8, 2024

Alexander Loitzl	Florian Zuleger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Alexander Loitzl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Mai 2024

Alexander Loitzl

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First, I want to thank the two people who made it possible for me to work on CompCert, a project I have grown fond of during the past year. Florian Zuleger, who welcomed a new topic and trusted me to overcome the associated challenges. Christian Ferdinand, who first gave me the opportunity to work on CompCert during an internship at AbsInt.

This thesis would not have been possible without the help of Bernhard Schommer, who gave me helpful insights into the CompCert development and who I could turn to when I was stuck. I would also like to thank Michael Schmidt, Christoph Cullmann and Christoph Mallon for the help with the testing environment at AbsInt, as well as discussions about CompCert and register allocation.

Most importantly, I would like to thank my parents for their continuous support throughout my studies.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Softwarekorrektheit ist essenziell in sicherheitskritischen Feldern wie der Luftfahrt und der Automobilindustrie. Hierbei genügt es nicht Verifikation nur auf dem Quellcode auszuführen, sondern es muss auch sicher gestellt werden, dass der genutzte Compiler keine Fehler verursacht. Anders als gewöhnliche Compiler schließt der formal verifizierte CompCert C Compiler solche Fehler aus, da er einen maschinengestützten Korrektheitsbeweis besitzt. Dieser Beweis garantiert korrekte Übersetzung des Quellcodes in die modellierte Assembly Sprache der Zielarchitektur.

Wir präsentieren CompCert^p, eine Erweiterung des CompCert Compilers welche Registerpaare modelliert. Registerpaare werden von 32-bit Prozessoren verwendet um 64-bit Gleitkommaarithmetik zu unterstützen. Zwei Hardwareregister können dabei zu einem Paar zusammengeschlossen werden, welches als Operand von Instruktionen verwendet werden kann.

CompCert^p unterstützt Registerpaare im Backend und modelliert die korrekte Gleitkommasemantik der Arm Architektur. Dadurch können wir die Aufrufkonvention (*calling convention*) für Gleitkommaargumente im bewiesenen Teil des Compilers unterstützen und sind kompatibel mit Bibliotheken, welche mit anderen Compilern übersetzt wurden. Außerdem können wir auf einen unverifizierten Teil von CompCert verzichten und erhöhen das Vertrauen in CompCert^{p}s Korrektheitsbeweis.

Wir beweisen CompCert^{p}s Korrektheit für alle Architekturen die CompCert unterstützt, und zeigen, dass CompCert^p mit leicht längerer Übersetzungszeit entweder kleineren oder vergleichbar großen Code generiert. Wir evaluieren CompCert^p auf bekannten Benchmarks und generierten Codebeispielen und erzielen bis zu 10% kleineren Code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Software correctness is crucial in safety-critical fields like aviation and the automotive industry. Performing verification only at the source-code level does not provide sufficient guarantees as the compiler might introduce bugs during the translation to machine-code. Unlike ordinary compilers, the CompCert formally-verified C compiler ensures correct translation to the target's modeled assembly language, therefore making additional verification redundant.

We present CompCert^p, an extension of the CompCert compiler that models register pairs, a feature found in 32-bit processors to implement double-precision floating-point arithmetic. Two hardware registers are grouped into a register pair that can be used as an operand of machine instructions to pass 64-bit values.

By supporting register pairs in the backend of the compiler, we can correctly model the floating-point semantics of the compiler's Arm target. This allows us to implement the correct calling conventions for floating-point arguments and be compatible with libraries compiled with different compilers. Moreover, we can omit one of the unverified passes of CompCert increasing the trust in CompCert^p's correctness proof.

We adapt the proofs for all supported architectures of CompCert and show that CompCert^p either improves on CompCert or performs similarly in terms of code size at the cost of a slight increase in compile time. For the Arm target, we evaluate CompCert^p on well-known benchmark suites and tests generated by a fuzzer, showing an improvement of up to 10% in code size.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

1.1 Motivation

Software has become an integral part of our everyday life. Not only does it surround us in the consumer electronics we all have gotten used to, but over the past decades it plays an increasingly important role in safety-critical fields like aviation and the automotive industry. For applications where bugs are just a nuisance to the end user, software testing is relied upon to ensure functionality. In safety-critical applications, where bugs can lead to catastrophic events, means for verification are employed throughout the entire development process.

Safety standards like the DO-178C [RTCA11] for avionics dictate verification at multiple development stages, in particular on the source code and machine code level. This redundant verification effort highlights the safety-critical industry's awareness that bugs might be introduced during compilation. In other fields, many software developers treat compilers as a black box, assuming that the behavior of the source code is reflected by the output machine code. In reality, compilers are highly complex software systems and bugs are regularly found and fixed [SLZS16; YCER11]. To minimize the risks of bugs silently introduced by the compiler (miscompilation), either an additional verification at the machine-code level is performed or a non-optimizing compiler is used to establish a one-to-one correspondence between the source code and the machine code.

This additional cost of verification effort or execution time is addressed by CompCert [Ler09b], a formally verified, optimizing C compiler. It is developed using Coq, an interactive theorem prover, to provide a proof of semantic preservation between a C-like intermediate language and the target architecture's assembly code. In other words, the behavior of the code output by the compiler corresponds to the behavior of the input program as prescribed by the C semantics. The proof covers all optimizations and non-trivial translation passes and therefore, unlike for other compilers, bugs usually do not affect these stages of the translation [YCER11].

CompCert is not exempt from the possibility of bugs and there are two main sources. First, a mistake in the modeled semantics of either C or the assembly languages can lead to miscompilation, even though a proof of correctness is established. Second, for some constructs or target architectures, unverified OCaml code is used to modify the final output before assembling to machine code. Both of these sources are part of the so-called Trusted Computing Base (TCB) of CompCert which also includes all software for which correctness is assumed. A detailed treatment of the TCB is given in [MB22]. This work aims to increase the trust in CompCert, by refining the semantic model of a target architecture and reducing the TCB.

1.2 Problem Statement

One of the key engineering goals of CompCert is to reduce the proof burden, especially redundancy in the architecture-specific reasoning. To overcome this issue, a common interface, shared by all architectures, is used to establish the properties required for the semantic preservation proof. Adopting such a shared view of the different architectures can also lead to discrepancies between the individual semantic models and the actual target language. Yet, the correctness of the assembly semantics can only be verified by human reviewers and is part of the TCB. The gap between the semantic representation and the actual target language can on the one hand lead to inefficient code generation, while in the worst case, unverified code is required to modify the output of the compiler. One such issue we identified is that of aliasing registers, two registers where writing to one invalidates the other.

CompCert currently works under the assumption that all registers are disjoint, hence, writing to a single register leaves all others unchanged. This property is enforced by the shared semantic model of the register file, yet it is not necessarily true for modern architectures. The 32-bit Arm floating-point coprocessor and the TriCore Aurix TC4x architecture use aligned pairs of adjacent registers to hold 64-bit values. Both architectures are prominently used in the automotive industry [Arma; Inf] and are desirable targets for CompCert. The floating-point support for Arm is limited: only half of the floating-point registers can be used by the compiler and it requires unverified code to conform to the Application Binary Interface (ABI). The TriCore architecture is currently not supported by CompCert but is planned for a future release. Unlike Arm, it does not have a designated set of floating-point registers and all computations use a shared set of data registers. A similar workaround is therefore not possible, as using only half of the registers for all computations leads to inadequate code generation.

1.3 Contributions

By adding support for register pairs in CompCert we reduce the TCB, generate more efficient code and ease the integration of new architectures in the future. Concretely, the key contributions of this thesis are:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1. We improve the semantic model of the Arm target by modeling all available registers and the correct calling conventions as prescribed by the ABI. These changes allow us to omit unverified modifications of the output after code generation.

2. We adapt the intermediate languages in CompCert's backend to handle register pairs. We include these changes in all existing targets to either take advantage of the new feature or to work as before.

3. To handle the new feature in the backend, we implement a register allocation algorithm similar to that described in [SRH04]. We adapt the validator described in [RL10] to handle register pairs.

4. All changes described above only affect the backend of the compiler. In particular, we do not require any changes to the memory model or value representation. Hence, the changes can be integrated into the CompCert compiler as is, and do not affect other projects relying on the CompCert C semantics.

5. We perform extensive benchmarks of our changes. We record changes in compile time due to the new register allocation algorithm and the size of the compiled executable. We test our contributions on standard compiler benchmarks and industry code.

1.4 Structure of the Thesis

In Chapter 2 we briefly review the properties of the two processor architectures that motivated our changes. We go on to give an overview of the CompCert compiler, reviewing the implementation and the high-level proof idea. Those components directly affected by our changes will be treated in more detail, in particular, register allocation.

In Chapter 3 we show the limitation of CompCert by stepping through a concrete example. We go on to discuss our design choices for the pair representation in both the register allocator and the backend of the compiler.

In Chapter 4 we discuss the implementation of our contributions. We highlight how our design choices affected the correctness proofs by looking at two translation steps in more detail.

In Chapter 5 we evaluate our changes by performing benchmarks and putting our contributions into the context of related work.

Finally, in Chapter 6 we sum up our contributions and discuss limitations and future directions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Preliminaries

Before we can detail and motivate the contributions of this thesis we first need to provide some background on the specific features of the affected Arm and TriCore architectures. We go on to review CompCert's internals, focusing on those parts that are affected by our contributions. We pay special attention to register allocation and give a detailed account of the allocation algorithm in CompCert.

2.1 Notational Conventions

In C, 64-bit and 32-bit floating-point numbers have types `double` and `float`, respectively. In CompCert, 64-bit floating-point numbers have type `float` and 32-bit floating-point numbers have type `single`. To use consistent names and avoid confusion, we use the type `double` for 64-bit floating-point numbers and the type `single` for 32-bit floating-point numbers. We use the same terminology in all pseudocode examples in this thesis, changing the names of types, functions and constructors where necessary.

We present examples in C, several low-level intermediate languages of CompCert and Arm assembly. Pseudocode showcasing our development is presented either in Gallina, the specification language of the Coq theorem prover, or OCaml. We do not expect any prior knowledge in any of the specific languages but assume some familiarity with assembly code and functional as well as imperative languages.

We sometimes mix source code and a more mathematical notation. To describe the result of a potentially failing computation (OCaml's option type), we write $\lfloor x \rfloor$ to denote successfully returning x and $\top$ to denote failure. For a map $x : Y \to Z$ we write $x(y)$ for accessing the map and $x[y \leftarrow z]$ for the map in which y is updated to z.

2.2 Architectures & Registers

In the following, we give a brief overview of the relevant parts of the Arm and TriCore architectures. We will focus on the layout of the register files and for the Arm architecture also detail the calling conventions.

2.2.1 Arm

CompCert supports Armv8 in 64-bit mode and Armv8, v7 and v6 in 32-bit mode. For the 32-bit architectures, it supports the additional VFPv2 (Armv6) and VFPv3-D16 floating-point coprocessors. All 32-bit Arm architectures are supported in little-endian or big-endian mode. The contributions of this thesis affect only the 32-bit architectures with floating-point support, hence, in the rest of this thesis we just use the shorthand Arm when we refer to those architectures.

Register Files

Arm's registers are split into core registers and extension registers. The processor has

- thirteen 32-bit General purpose registers (GPRs) `R0-R12`,
- three 32-bit special use registers `SP`, `LR` and `PC`, which can also be referred to as `R13-R15`
- sixteen 64-bit GPRs that can be seen as:
  - thirty-two single-precision GPRs `S0-S31`, or
  - sixteen double-precision GPRs `D0-D15`.

In Figure **??** below, we illustrate the two register files, the core registers on the left and the extension registers on the right. The sixteen double-precision registers are aligned pairs of single-precision registers. For $0 \leq \texttt{n} \leq 15$, the register `D<n>` is made up of the two registers `S<2n>` and `S<2n+1>`. The least significant half of a value stored in `D<n>` is in `S<2n>`.

We view the register files as split into three different classes. The class of integer registers R, the class of single-precision registers S and the class of double-precision registers D. Registers in D are what we refer to as *register pairs* and may alias with registers from S. If two registers alias, assigning to one invalidates the value in the other register. The register `D<n>` aliases with `S<2n>` and `S<2n+1>`.

Calling Conventions

The GPRs are split into caller-save and callee-save registers. The registers `R4-R11` and `S16-S31` are the callee-save registers. If a function uses these registers it is responsible for restoring the values they contained before the function was called. The other GPRs

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

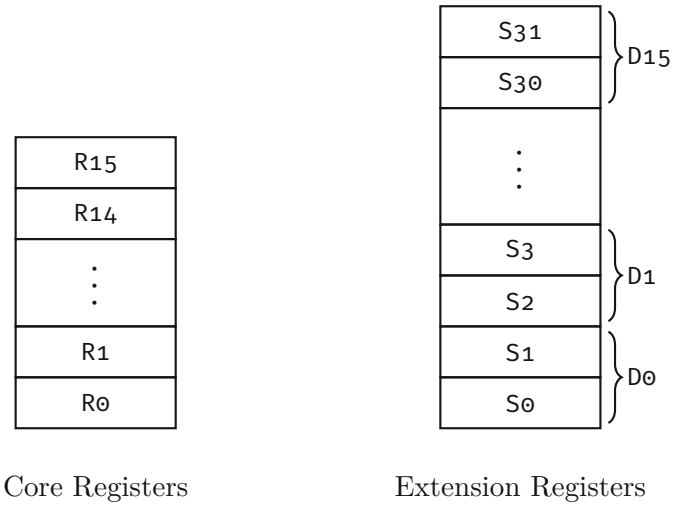


Figure 2.1: Arm register files

are the caller-save registers. The calling function is responsible for preserving their value and they are used for argument passing.

Without the floating-point coprocessor, parameters are passed in registers `R0-R3` and any additional parameters on the stack. This also applies to variadic functions where the number of formal parameters is not fixed. For non-variadic functions, floating-point arguments are passed in registers `S0-S15` (`D0-D8`). Starting with `S0`, the arguments are passed in the next available register respecting alignment constraints of the double-precision registers. If a register is skipped to respect the alignment constraints of an argument, it is used for the next argument that fits into the register. Consider the example of a simple C function f below.

```
          R0        S0       d1/(S3,S2)      S1
f( int i, single f1, double d1 , single f2)
```

The parameters of the function are annotated with the registers that their argument will be passed in. In particular, notice how the second single-precision parameter `f2` is passed in register `S1`, which was skipped to pass the third parameter `d1`.

2.2.2 TriCore

CompCert currently does not support the TriCore architecture but is planned to in a future release. The TriCore architecture has 32 GPRs split into address registers `A0-A15` and data registers `D0-D15`. Similar to the Arm floating-point coprocessor, aligned pairs of both data and address registers can be addressed. In Figure **??**, we show the register files. Function calls are handled by special instructions automatically storing and reloading the callee-save registers and we omit giving the details.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

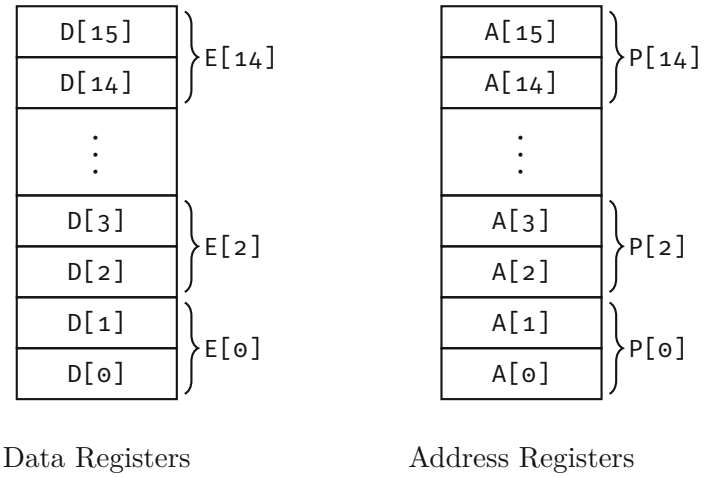


Figure 2.2: Tricore register files

2.3 CompCert

The CompCert project was initiated in 2005 and since then has been a prominent basis for low-level systems research and compiler verification. The main contributions are the formally verified, industrial-strength compiler CompCert and formal semantics for the C language. Throughout the years CompCert has seen many extensions [BBGH+19; BDP14; SCKK+19], has been integrated into larger toolchains [App11; AAMP+17] and its memory model has inspired others [JJKD17; SLKM+21]. CompCert has also seen industry adoption in safety-critical fields [KBWS+18], and has been shown to improve code size and worst-case execution time compared to previously used compilers [BBFL+12; KBWS+18]. For its contributions to research and industry, CompCert received the ACM Software Systems Award in 2021.

Below we give an overview of the structure of the compiler toolchain, focussing on the parts relevant for this thesis. CompCert, like a typical compiler, is split into a C-dependent frontend and a target-dependent backend. Additionally, it comes with rigorous semantics of all intermediate languages, required to reason about the correctness of the compiler. The review is based on multiple works detailing the components of CompCert [BDL06; LABS14; Ler09a; Ler23] and the latest development found online [LBDJ+].

2.3.1 Machine Model & Semantics

Reasoning about the behavior of a program requires a rigorous formulation of the semantics of the language it is written in. In CompCert's case, this requires formal semantics for C, the target's assembly language, and all intermediate languages of the compiler. Defining the semantics of imperative languages requires modeling the state

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Inductive val: Type :=
  | Vundef
  | Vint (n: int)
  | Vlong (n: int64)
  | Vdouble (f: float64)
  | Vsingle (f: float32)
  | Vptr (b: block) (ofs: ptrofs).
```

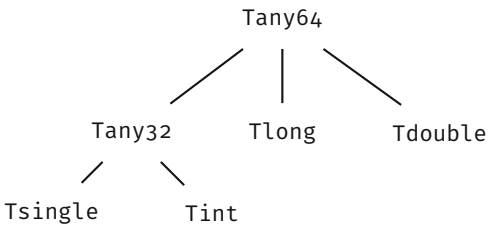


Figure 2.3: Values and types

they are manipulating. For the languages used in CompCert, we require models of the machine representation of values, memory, and registers.

Values

All languages operate on the `val` datatype given above in Figure 2.3. `Vundef` is an undefined value and for example, used to refer to the result of accessing uninitialized memory. The scalar types are a wrapper around the concrete representation of machine integers `n` or IEEE floating-point numbers `f`. Pointers are not represented at the bit-level but instead consist of a block identifier `b` and an offset `ofs` into the block.

The subtyping relation is given in Figure 2.3 on the right. The arithmetic values are typed intuitively and `Vundef` is of any type. Pointers either have type `Tlong` or `Tint`, depending on the target architecture. The types `Tany32` and `Tany64` are reserved for special operations on memory.

Memory

CompCert uses a single memory model for source, target and intermediate languages. A memory state is specified via an abstract datatype `mem` and consists of a collection of disjoint blocks. Addressing is performed via an integer offset into a block, allowing pointer arithmetic within a block.

Memory is modeled at the byte-level to capture low-level programming idioms. It comes with a more fine-grained notion of types capturing signedness and size. Memory types τ are passed as an argument to every memory access performing necessary type-conversions. Below we give an intuition of some memory operations, omitting those explicitly working on the byte-level (`store_bytes`, `load_bytes`, `drop_perm`).

- `alloc(m, lo, hi)` returns an updated memory and the address of the freshly allocated block of size `hi` − `lo`.
- `store(`τ`, m, b, ofs, v)` returns an updated memory with block `b` at `ofs` containing the value `v` encoded according to τ. A `store` can fail if the memory at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the given address is not writeable. We use the following notation for a `store`: $\mathtt{m}[(\mathtt{b}, \mathtt{ofs}) \leftarrow_{\tau} \mathtt{v}]$.

- `load(` τ`, m, b, ofs)` decodes and returns the value at `ofs` in block `b`. A `load` can fail if the address is not readable. We use the following notation for a `load`: $\mathtt{m}[(\mathtt{b}, \mathtt{ofs})]_{\tau}$.
- `free(m, b, lo, hi)` frees a memory quantity. A `free` can fail if an already freed block is freed again.

The abstract datatype specifies the behavior of the operations in a natural fashion. The example below states that after successfully storing a value encoded with τ', a load from the same location succeeds if loaded with a compatible type τ. We write $\tau \sim \tau'$ to denote compatibility between two types, e.g., the types of 32-bit signed and unsigned integers.

$$\text{If } \mathtt{m}[(\mathtt{b}, \mathtt{ofs}) \leftarrow_{\tau'} \mathtt{v}] = \lfloor \mathtt{m'} \rfloor \text{ and } \tau \sim \tau' \text{ then } \mathtt{m'}[(\mathtt{b}, \mathtt{ofs})]_{\tau} = \lfloor \mathtt{v} \rfloor.$$

Since memory is modeled at the byte level, a `load` can also succeed even if no previous `store` was performed on the exact address loaded from. We can for example load only one half of a 64-bit integer previously stored in a block b at ofs, by loading a 32-bit quantity from block b at $ofs + 4$. This is only possible for the arithmetic types as pointers are not modeled at the byte level. Moreover, CompCert also supports loads and stores using the types `Tany32` and `Tany64`. Like pointers, the values are not split into the individual bytes and any overlapping load will fail.

Register File

CompCert models the register file rs of its intermediate languages as a simple map $rs : Regs \rightarrow \mathtt{val}$ from the set of registers $Regs$ to values. Hence, all registers are disjoint and untyped giving us two convenient properties to reason about updates. First, assigning a value v to a register and reading back from r returns the same value v.

$$\forall r, v : \ rs[r \leftarrow v](r) = v.$$

Second, if two registers r and r' differ, setting the value of one leaves the value stored in the other unaffected.

$$\forall r, r', v : \ r \neq r' \Rightarrow rs[r' \leftarrow v](r) = rs(r).$$

Since registers are assumed to be disjoint Arm's register file cannot be modeled correctly. The employed workaround is that only the double-precision registers are used. This cuts the number of available registers for single-precision computations in half. Moreover, the calling conventions cannot be modeled correctly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2.3.2 Preprocessing & Compiler Frontend

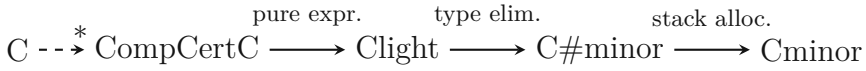


Figure 2.4: Translation steps from C to Cminor

The source language of CompCert is a large subset of C99 with some extensions from C11. The initial preprocessing and parsing contains multiple unverified steps (indicated by a dashed arrow in Figure 2.4). It translates the input program to the CompCertC language, the entry point to the verified part of the compiler. To increase trust in this unverified pass, one can optionally output the resulting CompCertC program. Static analysis can then be performed on the preprocessed CompCertC program, instead of the C source code.

The next translation step to Clight pulls side effects out of expression, inserting appropriate statements capturing the semantics of the side effects. The final two translation steps resolve type-dependent behavior, like overloaded operators, and identify those variables whose address is never taken. Local variables of a function whose address is not taken (temporaries) are subject to register allocation, while the others are allocated on the stack. The resulting intermediate language Cminor is a target-independent, untyped, low-level language.

2.3.3 Compiler Backend, Assembling & Linking

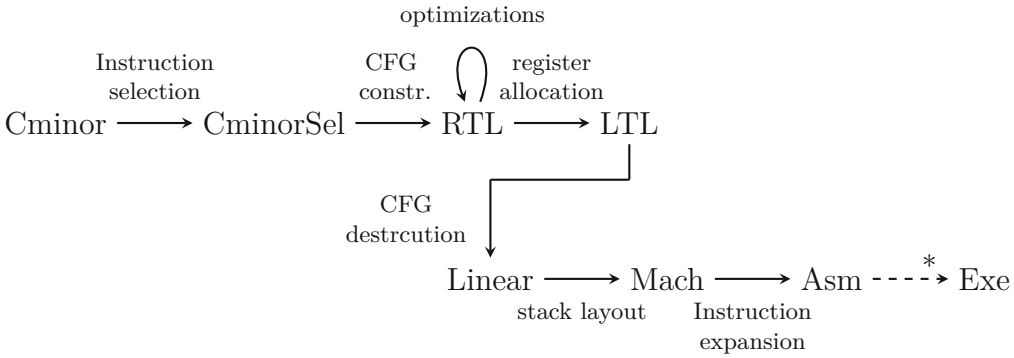


Figure 2.5: Translation steps from Cminor to the executable

The first translation of the backend is to CminorSel, a variant of Cminor in which processor-dependent instructions are selected for expressions. Simple constant propagation and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

arithmetic identities are used to minimize the number of required instructions. The next translation step to RTL introduces fresh temporaries for intermediate results and the program is transformed into a Control-flow graph (CFG). RTL is the target language of almost all optimizations of CompCert and is the last intermediate language unaffected by our contributions.

To get an intuition for the intermediate languages and translation steps we show the signatures of selected RTL and LTL instructions below in Figures 2.6 and 2.7. The RTL

```
Inductive instruction: Type :=
  | Iop (op: operation) (r⃗: list reg) (r: reg) (s: node)
  | Iload (τ: memory_chunk) (addr: addressing) (r⃗: list reg) (r: reg) (s: node)
  | Icall (sig: signature) (ros: reg + ident) (r⃗: list reg) (r: reg) (s: node).
```

Figure 2.6: Selected RTL instructions

instructions operate on temporaries (type `reg`) and a map `rs`, from temporaries to their stored values, is part of the program state. Each instruction takes a list of arguments $\vec{r}$ and returns a result in `r`. The nodes of the CFG are single instructions, hence each instruction has a successor `s`. The semantics of the instructions follow a common pattern. The arguments are evaluated using the map `rs`. The values are used in an arithmetic operation `op`, used to compute the address using the addressing mode `addr`, or passed as arguments to a function call. The result is then stored in `rs` at the specified destination `r`.

The translation between LTL and RTL is register allocation and assigns machine registers (type `mreg`) and stack slots to temporaries. An LTL execution state has a location map `ls` from stack slots and machine registers to values. For the selected LTL instructions given below, only machine registers are allowed to pass as arguments and store results in.

```
Inductive instruction: Type :=
  | Lop (op: operation) (r⃗: list mreg) (r: mreg)
  | Lload (τ: memory_chunk) (addr: addressing) (r⃗: list mreg) (r: mreg)
  | Lcall (sg: signature) (ros: mreg + ident)
  | Lbranch (s: node).
```

Figure 2.7: Selected LTL instructions

The instructions are syntactically and semantically similar to the RTL instructions. They do not have a successor, as in LTL a basic block in the CFG can have multiple instructions and is terminated with the newly introduced `Lbranch` instruction. The `Lload` and `Lop` instructions directly correspond to their counterpart in RTL, using `ls` to evaluate the registers. The `Lcall` instruction does not have any arguments or destination as the argument values are passed in machine registers according to the signature of the function.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The next translation step to Linear destructs the CFG and the step to Mach, makes the stack layout of a function explicit, hence called the Stacking pass. Up until LTL, stack accesses are modeled using the location map. In Mach, the stack is treated as part of the memory. Another responsibility of the Stacking pass is to insert instructions to preserve callee-save registers across a function call. Previously, callee-save registers were automatically restored and built into the semantics of calls. In the Stacking pass, the callee-save registers that are used by a function are determined and instructions to store them on the stack are inserted in the function prologue. Symmetric instructions to reload the registers are inserted in the function epilogue.

The last formally proven translation step is to Asm, the targets's assembly semantics. The final steps of generating the executable, are once again unverified and an off-the-shelf assembler is used. On some architectures, there are additional unverified passes before assembling to fix potential discrepancies between the modeled semantics and the actual target.

2.3.4 Correctness Proof

CompCert is implemented using the Coq interactive theorem prover and machine-checked proofs are carried out directly on its source code. In the proven correct parts of the compiler, this rules out any miscompilation with near-certainty. The proof of correctness relates the source and output program via the notion of semantic preservation.

Definition 2.3.1 (Semantic Preservation [Ler23])
For all source programs S and compiler-generated code C,
if the compiler, applied to the source S, produces the code C,
without reporting a compile-time error,
then the observable behavior of C improves one of the allowed observable behaviors of S.

Note that the compiler need not output code for any particular source code S. CompCert may fail at compile-time if the input is malformed. It may also fail on well-formed input, e.g., if it cannot allocate a function without exceeding the maximum stack size. Moreover, the compiler is allowed to *refine* the behavior of S. The C expression evaluation is non-deterministic and the compiler can choose one of the allowed execution orders. It is also allowed to improve on the behavior, i.e., giving a program with undefined behavior meaning. This is necessary to perform optimizations like dead-code elimination, which might remove sources of undefined behavior.

Semantic preservation is established between any two intermediate languages and the proofs are composed to the main theorem of semantic preservation between CompCertC and the target's assembly semantics Asm. To prove the correctness of the individual translations, two approaches are used. We write $S \approx C$ to denote semantic preservation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Definition 2.3.2 (Verified Compiler [Ler09a])
A compiler is said to be verified if it is accompanied with a formal proof of the following property:

$$\forall S, C: \; Comp(S) = OK(C) \implies S \approx C$$

Adopting the above terminology, we can view each translation step of CompCert as a single compiler between two intermediate languages. Most translation passes are a *Verified Compiler* implemented in Coq requiring each translation step to be reasoned about. Another approach, e.g., used for register allocation, is that of a *Verified Validator* defined below.

Definition 2.3.3 (Verified Validator [Ler09a])
A validator *Validate* is said to be verified if it is accompanied with a formal proof of the following property

$$\forall S, C: \; Validate(S, C) = true \implies S \approx C$$

The compiler itself does not need to be verified and only the result is inspected. The validator itself is proven to only accept the program C if it preserves the semantics of S. This can be convenient for very complex translations since one does not have to reason about every compilation step.

2.3.5 Pairs of Registers in CompCert

CompCert supports 64-bit integer arithmetic on its 32-bit target architectures where the operations have no corresponding machine instructions. Instead, library functions with axiomatized semantics are relied on to offer full support. Since 32-bit registers cannot hold values of type `Tlong`, values are split up into two registers. Semantically, this is done by the functions `makelong`, `lowordoflong`, and `hiwordoflong`. The functions are only defined on values `Vlong n`, accessing the bit representation `n` of the value. The semantics of 64-bit arithmetic is therefore only defined on values `Vlong n` and otherwise `Vundef`. This is no different from the arithmetic machine operations returning `Vundef` for those values that are not supported.

Similarly, values of type `Tlong` can be used as function arguments and are, according to the target's calling conventions, potentially passed in two registers. This is captured by the type `rpair mreg`, containing either one or two registers.

The difference to the register pairs we refer to in the course of this thesis is that they are only an abstraction adopted by the compiler. Independent machine registers can be grouped into a pair, capturing that the compiler used them to hold two halves of a value. They are deliberately limited in use, only available where the compiler uses them for 64-bit integer arithmetic. The register pairs we support with our contributions are a notion of the target's machine semantics and can be used as arguments to machine

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

instructions. This requires adopting pairs at the level of the machine semantics and using them in all places where only single registers are supported so far.

2.4 Register Allocation

A register allocator takes a program operating on an unbounded number of temporaries and transforms it into a semantically equivalent program operating on a finite number of machine registers. Temporaries are those variables of a program whose address is never taken and therefore do not need to be represented in memory. An optimizing compiler relies on the ability to create temporaries when needed, e.g., to hold a common sub-expression to avoid recomputation. Hence, good register allocation is crucial for an optimizing compiler.

CompCert uses a graph-coloring allocator, an approach available in many modern compilers [TMKF+19; GNU; GHC]. In the following, we give a detailed treatment of graph-coloring allocation and the concrete algorithm used in CompCert.

2.4.1 Graph-coloring Register Allocation

Graph-coloring register allocation is a global register allocation approach. Rather than assigning machine registers per basic block, allocation is performed per function. At the heart of a graph-coloring register allocator lies the interference graph. The nodes are comprised of temporaries and the interference edges connect two temporaries that are live at the same time, hence cannot be mapped to the same machine register. This gives us an immediate reduction to the coloring problem by determining the number of colors K by the available machine registers. A successful coloring gives us an assignment of machine registers such that no two interfering temporaries are mapped to the same register.

In Figure 2.8 we compiled a simple code snippet with CompCert. For the sake of this example, we manually turned off optimizations during the allocation phase. In the resulting RTL code, each line is annotated above with the set of temporaries that are live before the instruction. The corresponding interference graph is constructed using the liveness information of the RTL program. Two temporaries are connected by an interference edge, the solid lines, if one is live after an assignment to the other. Consider the two temporaries $x1$ and $x3$. In line 6 of the RTL code is an assignment to $x3$, but $x1$ is live after the instruction as it is used in line 8. This prohibits both $x1$ and $x3$ from being assigned to the same machine register. The dashed lines record moves between two registers, e.g., line 2. We can also have precolored vertices in the graph such as in this case `R0`. It is used to enforce the calling conventions, passing the parameter of f in register `R0`.

Casting register allocation as a coloring problem raises two issues. First, unlike a coloring problem, we cannot deem a graph uncolorable or find the smallest number of necessary colors instead. Second, graph coloring is NP-complete and we cannot afford to naively

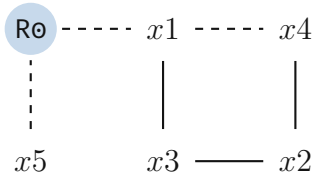
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
1 int f (int a) {
2   int d = a;
3   int b = 1;
4   int c = b + a;
5   d = c + b;
6   return d + c;
7 }
```

(a) C code

```
1  f() {
2    x1 = R0;
3  {x1}
4    x4 = x1;
5  {x1}
6    x3 = 1;
7  {x1, x3}
8    x2 = x3 + x1;
9  {x2, x3}
10   x4 = x2 + x3;
11 {x2, x4}
12   x5 = x4 + x2;
13 {x5}
14   R0 = x5; return R0
15 }
```

(b) RTL code



(c) Interference Graph

Figure 2.8: Simple C-code and corresponding interference graph

search for an optimal solution. To see how these issues are addressed we perform register allocation on our example graph using Chaitin's Algorithm [CACC+81; Cha82], the first implementation of a graph-coloring register allocator. The algorithm is separated into 4 phases detailed below, and the effects on the graph are illustrated in Figure 2.9.

1. ***Coalescing:*** The *Coalescing* phase combines any two non-interfering nodes that participate in a move. When combining two nodes, we take the union of their neighbors. In our example, we therefore combine vertices $x1$, $x4$, $x5$ with `R0`, keeping the color. If any nodes are coalesced, the graph is rebuilt, potentially leading to new opportunities for further coalescing.

2. ***Simplify:*** The *Simplify* phase removes vertices from the graph based on the following observation: If a vertice v has less than K neighbors, a graph G is K-colorable if and only if $G \setminus \{v\}$ is K-colorable. This simple criterion is used to remove vertices from the graph, pushing them onto a stack, until no more low-degree ($< K$) vertices are in the graph. In our example, we can repeat this until we end up with the empty graph.

3. ***Spilling:*** If the above strategy gets stuck we give up on coloring the graph. We pick a node in the graph and rather than finding a register for it, we decide to store it on the stack instead. We insert spill-code into the program that loads and stores the temporary before and after every use. If we need to spill a temporary we start over on the new program. In our example, we do not need to spill any temporaries.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4. ***Select:*** The *Select* phase is responsible for the coloring. It pops temporaries off the stack assigning a color for each. Since we traverse the temporaries in the reverse order than they were removed in, we are guaranteed that the vertice has less than K neighbors and therefore is colorable.

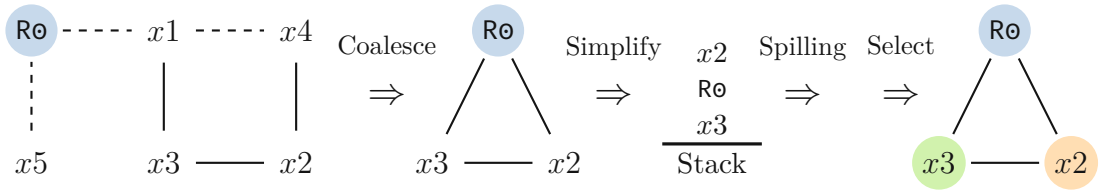


Figure 2.9: Coloring using Chaitin's algorithm with $K = 3$

2.5 Register Allocation in CompCert

The register allocation in CompCert is an implementation of Iterated Register Coalescing (IRC) [GA96]. Similar to Chaitin's algorithm, it builds an interference graph and manipulates it in several phases, pictured below. The *Coalescing* phase and the *Simplify* phase are interleaved in a loop. Furthermore, it uses a different strategy for *Coalescing* and *Spilling*. We discuss the changes and new phases below.

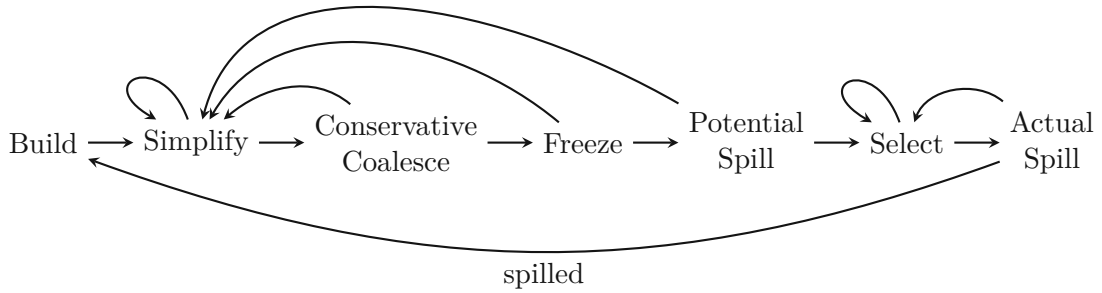


Figure 2.10: Phases of IRC [GA96]

1. ***Simplify:*** While Chaitin's algorithm removes all low-degree nodes, IRC only removes low-degree nodes that do not participate in any move.

2. ***Coalescing:*** IRC relies on the *Conservative coalescing* strategy introduced by Briggs et al. [BCT94]. It combines nodes only if the resulting node has less than K neighbors of high-degree ($\geq K$). This guarantees that a graph colorable with K colors does not turn uncolorable by coalescing. Consider the example of Chaitin's

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Algorithm in Figure 2.9. The original graph was two-colorable while the graph after coalescing is not. If we try to color it with $K = 2$, Chaitin's algorithm inserts unnecessary spill-code. Since *Coalescing* and *Simplify* create opportunities for further progress for one another they are repeated in a loop.

3. ***Freeze:*** If neither *Simplify* nor *Conservative Coalesce* can make any progress, we give up on coalescing a move between two low-degree nodes. This will lead to further opportunities for *Simplify*.

4. ***Spilling:*** Based on *Optimistic Coloring* [BCT94], spilling decisions are shifted from the *Simplify* phase to the *Select* phase. Unlike Chaitin's Algorithm which spills a node if no progress can be made during *Simplify*, IRC picks a node, pushes it onto the stack and continues with *Simplify*. Now, during *Select*, we are not guaranteed to find a color. If no color is available we spill the node and rebuild the graph on the modified program.

In Figure 2.11 below, we color our example interference graph with IRC, but this time with $K = 2$. Since all low-degree vertices are part of a move, we cannot make any progress in *Simplify*. The initial *Coalesce*, unlike Chaitin's Algorithm, will not combine all move-related nodes as the resulting node would have degree two. After *freezing* the move, *Simplify* will once again be able to remove all nodes after which we can find a valid coloring in the *Select* phase.

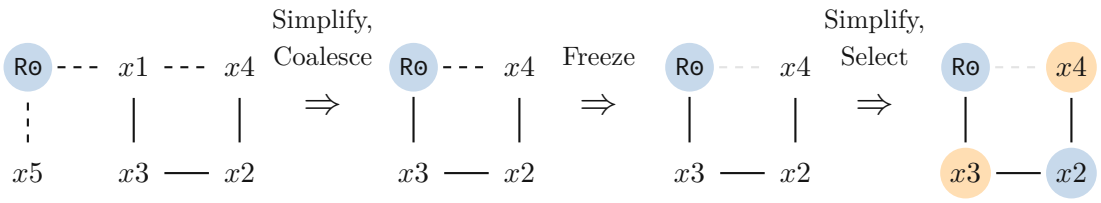


Figure 2.11: Coloring using IRC with $K = 2$

2.5.1 Allocation Validation

The allocation validation algorithm forms the translation step from RTL to LTL and follows the principle of a *Verified Validator*. It relies on an untrusted register allocator to produce LTL code and checks if the result is semantics-preserving. Similar to register allocation, validation is performed function by function. First, the allocator performs a structural check, verifying that the two programs are similar. This deliberately limits the scope of the validator and reduces the proof effort. Second, a data-flow analysis relates RTL temporaries with LTL locations, ensuring that they contain the same value for all executions of the program. Below we give an intuition of the validation algorithm based on [RL10], focusing on those parts that are affected by our changes. For a more in-depth description, we refer to the original description in [RL10] and the current development [LBDJ+].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Structural Checks

After receiving the LTL code from the allocator, the validator establishes a correspondence between RTL instructions and LTL basic blocks. The register allocator needs to be able to insert new instructions, e.g., *Spilling*, or remove instructions, e.g., *Coalescing*. The validator imposes two limits on the program transformations the allocator may perform. First, instructions without a corresponding one in the RTL code can only be moves, including stack accesses. Second, every RTL instruction needs a corresponding LTL instruction, therefore forcing the allocator to replace an instruction with a `nop` instead of removing it. The LTL code output by the allocator is only accepted if each RTL instruction is related to one LTL basic block containing a matching LTL instruction potentially preceded by moves between locations. A matching LTL instruction can either be a `nop` or an equivalent LTL instruction using locations instead of temporaries.

Data-Flow Analysis

The second part of validation is a backward data-flow analysis, statically ensuring that the same values flow through the RTL and LTL programs. It manipulates a set of equations E relating RTL temporaries with LTL locations. An equation between a temporary x and a location l can take three shapes:

1. $x =_F l$, equating the value stored in x and l (F stands for full),
2. $x =_L l$, equating the value stored in l to the lower half of that stored in x,
3. $x =_H l$, equating the value stored in l to the upper half of that stored in x.

Equations typically are of the first form, establishing a one-to-one relation between a temporary and a location. The other two equation kinds are used to support 64-bit integer arithmetic on 32-bit machines. As mentioned in Section 2.3.5, values of type `Tlong` may be split up into two machine registers. Each half is then related to the temporary that was used to hold the entire value in the RTL code.

The analysis uses standard fixpoint iteration to compute the set of equations that need to hold at the entry point of a function f. The result is checked for compatibility against the set of equations relating f's parameters with the machine registers that are used to pass the arguments in the LTL code.

To prepare the presentation of parts of the transfer function, we first define some operations on equation sets. A pair (x, l) is compatible with a set of equations E, if all equations either relate x and l or involve neither of the two. We give the formal definition below, where $l \perp l'$ asserts that the two locations do not overlap.

$$(x, l) \perp E \stackrel{def}{=} \forall (x' =_\kappa l') \in E : \ (x' = x \wedge l' = l \wedge \kappa = F) \vee (x' \neq x \wedge l' \perp l).$$

Compatibility between a pair (x, l) and a set E is checked routinely and the equation sets are therefore implemented redundantly, as AVL trees. One is ordered first by temporaries, the other first by locations allowing us to quickly find all equations that contain x or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

locations overlapping with l. Next, we define the substitution of a location. It simulates the effect of an assignment of l_s to l_d on a set of equations E. If E contains an equation with a location l' partially overlapping with l_d, denoted $l' \# l_d$, the set of equations becomes unsatisfiable, denoted by $\top$.

$$E[l_d \leftarrow l_s] \stackrel{def}{=} \top \text{ if there exists } (x = l') \in E \text{ s.t. } l' \# l_d.$$
$$E[l_d \leftarrow l_s] \stackrel{def}{=} \{(x =_\kappa l_s \mid (x =_\kappa l_d) \in E\} \cup \{(x =_\kappa l) \mid (x =_\kappa l) \in E \wedge l \perp l_d\})\}.$$

Below we give a pseudocode example of the transfer function used in the data flow analysis. Given an equation set E, and two related pieces of RTL and LTL code it computes the set of equations that has to hold before execution, such that E holds after execution. The first case is the simple case of two arithmetic operations being

```
transfer_instr instr bblock E :=
  match instr, bblock with
  | Iop _ x⃗ x _, Lop _ l⃗ l _ =>
      if (x,l)⊥E then E \ {x = l} ∪ {x⃗ = l⃗} else ⊤
  | _, (Lop mov l_d l_s) :: bblock => transfer_instr instr bblock E[l_d ← l_s]
  end.
```

Figure 2.12: Parts of the transfer function

related. Intuitively, if we can establish the equality of all arguments before executing the operation, we know that the results contain the same value. Additionally, we need to check the compatibility to rule out invalidating any other equations by the assignment. The second case corresponds to a move inserted by the allocator. Any equation that involves l_d can only be satisfied by the RTL code if it holds with l_s before the assignment.

Proving Soundness

The soundness proof guarantees that the validator only accepts the LTL code if it preserves the semantics of the RTL code. The proof maintains and uses the notion of equation satisfaction defined below. It relates an RTL register map rs to an LTL location map ls.

$$rs, ls \models E \stackrel{def}{=} \forall (x =_\kappa l) \in E,\ rs(x) =_\kappa ls(x).$$

The soundness proof proceeds by forward simulation, stepping through the RTL and LTL program maintaining equation satisfaction between the state of the register map and the location map. In other words, we prove that the semantics of executing an instruction is reflected by an update of the equation set according to the transfer function.

Below we state the lemmas corresponding to the two cases of the transfer function above. Note the symmetry between the lemmas and the transfer function. The transfer function is used in a backward data-flow analysis, while the lemmas are used in the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

forward simulation proof. Each Lemma relates the manipulation of equation sets with the equivalent operations on the register and location maps.

Lemma 2.5.1 lets us reason about the simultaneous assignment of two related values v and v' to a temporary x and a location l. If rs and ls satisfy the equation set without the equation relating x and y, and (x, y) is compatible with E, a simultaneous update of x in rs and l in ls satisfies E.

Lemma 2.5.1 (Simultaneous Assignment)
If $rs, ls \models E \setminus (x =_\kappa l)$, $(x, l) \perp E$, *and* $v =_\kappa v'$ *then* $rs[x \leftarrow v], ls[l \leftarrow v'] \models E$.

Lemma 2.5.2 corresponds to the second case of the transfer function. If rs and ls satisfy the equation set in which we replace all occurrences of l_d with l_s then rs and $ls[l_d \leftarrow l_s]$ satisfy E.

Lemma 2.5.2 (Inserted Move)
If $E[l_d \leftarrow l_s] \neq \top$ *and* $rs, ls \models E[l_d \leftarrow l_s]$ *then* $rs, ls[l_d \leftarrow ls(l_s)] \models E$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Design

In this chapter, we give an overview of CompCert^p, an extension of CompCert supporting register pairs. To motivate the changes, we step through the compilation of a simple example program, showing where CompCert^p improves on the current state of CompCert. After the motivation, we detail the scope of our changes and discuss our design choices.

3.1 Illustrating the issues via an example

We compile the simple C program in Figure 3.1. It calls the function `sum` which adds three floating-point numbers together. The parameters a and c are single-precision and their sum is explicitly cast to a double-precision value before adding b. The corresponding RTL code, on the right, closely resembles the C code and is the same for CompCert and CompCert^p.

```
double sum(single a, double b, single c){
  double d = (double) (a + c);
  return d + b;
}

int main(){
  sum (0.f, 1.0, 2.f);
  return 0;
}
```

```
sum(x5, x4, x6){
  x2 = x5 +fs x6;
  x1 = doubleofsingle(x2);
  x3 = x1 +f x4;
  return x3;
}

main(){
  x2 = 0.f; x3 = 1.; x4 = 2.f;
  x5 = "sum"(x2, x3, x4);
  x1 = 0;
  return x1;
}
```

Figure 3.1: Simple program summing up floating-point numbers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

The differences start after RTL, during register allocation, when assigning machine registers to the temporaries. In Figure 3.2 below, we show the two interference graphs of the `sum` function after coalescing. The interference graph of CompCert on the left uses floating-point registers `F<n>` treating single-precision and double-precision temporaries the same. Even coalescing between the two is possible as witnessed by the coalescing of `F0`, $x5$, and $x3$. CompCert^p on the other hand distinguishes between single-precision and double-precision temporaries and makes use of all of Arm's registers. A weighted directed graph is used to capture the effects between nodes of different register classes. The edge from `D1`/$x4$ to $x2$ has weight 2 as picking a double-precision machine-register blocks 2 candidates for any interfering single-precision node.

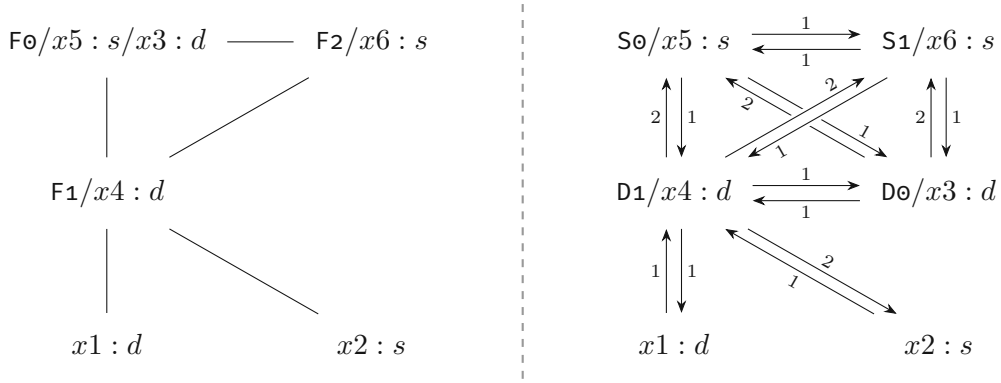


Figure 3.2: Interference Graphs of `sum`

The interference graph of CompCert (left) is undirected, while that of CompCert^p (right) is a directed weighted graph. We illustrate the effect of coalescing by listing all individual nodes separated by a "/". The type of a temporary is indicated by s for single-precision and d for double-precision.

```
1  sum() {
2    F0 = F0 +fs F2;
3    F0 = doubleofsingle(F0);
4    F0 = F0 +f F1;
5    return;
6  }
7
8  main() {
9    F0 = 0.f; F1 = 1.; F2 = 2.f;
10   call "sum";
11   R0 = 0;
12   return;
13 }
```

```
1  sum() {
2    S0 = S0 +fs S1;
3    D0 = doubleofsingle(S0);
4    D0 = D0 +f D1;
5    return;
6  }
7
8  main() {
9    S0 = 0.f; D1 = 1.; S1 = 2.f;
10   call "sum";
11   R0 = 0;
12   return;
13 }
```

Figure 3.3: Mach code of CompCert (left) and CompCert^p (right)

In Figure 3.3 is the corresponding Mach code. It closely resembles the RTL code,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

but temporaries have been replaced with machine registers. CompCert still uses the ambiguous `F<n>` registers rather than the hardware registers. The consequences of this representation are visible in the assembly output below.

```
1  sum:
2    vmov.f32 s4, s1
3            ⋮
4    vadd.f32 s0, s0, s4
5    vcvt.f64.f32 d0, s0
6    vadd.f64 d0, d0, d1
7            ⋮
8  main:
9            ⋮
10   vmov.f32 s0, #1.
11   vmov.f64 d1, #2.
12   vmov.f32 s4, #3.
13   vmov.f32 s0, s0
14   vmov.f32 s1, s4
15   bl  sum
16   mov  r0, #0
17           ⋮
```

```
1  sum:
2            ⋮
3    vadd.f32 s0, s0, s1
4    vcvt.f64.f32 d0, s0
5    vadd.f64 d0, d0, d1
6            ⋮
7  main:
8            ⋮
9    vmov.f32 s0, #1.
10   vmov.f64 d1, #2.
11   vmov.f32 s1, #2.
12   bl  sum
13   mov  r0, #0
14           ⋮
```

Figure 3.4: Assembly output of CompCert (left) and CompCert^p (right)

The output is stripped of all assembler directives and instructions dealing with function calls which are identical for CompCert and CompCert^p. The output of CompCert^p on the right is in a one-to-one correspondence with the Mach code. In the output of CompCert on the left three additional move instructions are highlighted in red. The insertion of the `vmov` instruction in line 13 was an oversight and has since been fixed[1]. The `vmov` instructions in lines 2 and 14 on the other hand are necessary. Since CompCert does not properly model all the machine registers of Arm, it cannot enforce the correct calling conventions. The `vmov` instructions copy values from the registers that are used for parameter passing according to the ABI to those used by the code output by CompCert. These instructions are inserted by an additional pass after compilation and are not captured by CompCert's proof.

3.2 Supporting Register Pairs in Allocation

The register allocator is inspired by the Generalized Register Allocator (GRA) introduced in [SRH04]. It is built on top of the existing implementation of IRC and exposes the concept of register classes to the allocator. For Arm, we give the allocator access to

[1] `https://github.com/AbsInt/CompCert/commit/ccb88a8e49fd28029e086a7b0855bbe0216e3bc0`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

the class of integer GPRs R, the class of single-precision GPRs S, and the class of double-precision GPRs D. As for GRA, we define the measure $worst(C, C')$ between any two classes C and C'. It captures how many candidates for a register in class C can be blocked if it interferes with a register of class C'. For Arm, we get $worst(S, D) = 2$ and $worst(D, S) = 1$. A single-precision register can only block one double-precision register, while a double-precision register blocks two single-precision registers. If registers of two classes cannot interfere, e.g., an integer register with a floating-point register, we leave $worst$ undefined. For all architecture except Arm, $worst(C, C') = 1$ if $C = C'$ and otherwise is undefined.

We use a weighted directed interference graph $G = (V, E, \omega)$ to capture the effect of allocating registers in multiple register classes. As usual, the vertices consist of the temporaries and the edges connect two interfering nodes. If two vertices u, v interfere they are connected by the edges (u, v) and (v, u). For any edge $e = (u, v)$, the associated weight $\omega(e) = worst(V, U)$ where U and V are the classes of u and v respectively. The degree $d(v)$ of a node v is the sum of the incoming edge weights:

$$d(v) = \sum_{e=(u,v)\in E} \omega(e).$$

For the *Simplify* phase, where low-degree nodes are removed from the graph, we use $worst$ to compute the degree based on the class of nodes. For the *Select* phase and *Coalescing*, we need to define aliasing between individual registers. During *Select* when assigning a machine register to a node, we check the neighbors that have already been assigned. When determining available registers, rather than checking for equality with neighbors, we check for aliasing. For *Coalescing*, we need to ensure that we do not create a shared lifetime between two aliasing machine registers.

Consider the two example graphs in Figure 3.5. Each graph has two single-precision temporaries $x1$, $x2$ and two double-precision temporaries $x3$, $x4$. The graph on the left is similar to that in the example of Section 3.1 and shows the case of an architecture with aligned pairs of single-precision registers. The only interesting case is the edge $(x4, x1)$ with weight 2. An assignment of a register to $x4$ blocks two candidates for $x1$. On the right side, we show the case of unaligned register pairs. The omission of alignment constraints leads to higher constraints between two interfering nodes. The most interesting case is the interference between the two double-precision registers $x4$ and $x3$ which has a weight of 3. A pair p aliases with itself, but can also alias with the two pairs which share one of the registers with p.

Figure 3.5: Interference graphs with aligned (left) and unaligned pairs (right)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.3 Supporting Register Pairs in CompCert

We strive for an easy integration of our contributions into CompCert and therefore refrain from adapting the semantics of the machine model. Since the memory model and the value representation are shared by all languages, adapting either of the two greatly increases the proof burden and potentially interferes with projects relying on CompCert's C semantics. We therefore considered two approaches, both restricted to the backend of the compiler.

1. **Modeling register aliasing:** Similar to the approach taken for the allocator, register classes are modeled and their aliasing is handled explicitly. Whenever a value is assigned to a register, all aliasing registers are set to `Vundef`. This approach is not restricted to pairs and allows support for arbitrary aliasing relationships between registers.

2. **Using optional register pairs:** This approach is restricted to register pairs. Aliasing is not modeled explicitly, but instead, we model the pair a register represents. For Arm, instead of using register `D<n>`, the pair `(S<2n+1>, S<2n>)` is used. Assigning a value to a pair requires the value to be split in two and each half is stored in the individual registers. Any aliasing is therefore handled implicitly as assigning a value to a register `S<n>` invalidates the value stored in any register pair containing `S<n>`.

We followed the second approach as strictly following the first approach leads to problems with the preservation of callee-save registers across function calls. A function needs to ensure that the state of each callee-save register is restored before returning. Since, on hardware, `D<n>` is the same location as `(S<2n+1>, S<2n>)`, callee-save registers potentially alias with one another. This prohibits modeling the preservation of all callee-save registers, as assigning a value to a register invalidates all aliasing registers.

In Figure 3.6 we illustrate the scope of our changes putting them into the context of the whole semantics used throughout compilation. The affected components and translation steps are highlighted in red. Pairs are introduced starting at LTL and resolved during Asm generation for the architectures not using pairs, leaving their semantics untouched. In the following, we give a more detailed account of the concrete changes to the affected intermediate languages and their translation steps, as well as the Arm assembly semantics.

3.3.1 Changes to Intermediate Languages & Translations

Components that stretch several intermediate languages, like the memory or arithmetic machine operations (Op), all operate on values. Specifics of an intermediate language, e.g., storing values in temporaries (RTL) or machine registers (LTL) are resolved in the semantics of the respective intermediate language. This allows us to handle pairs at the level of the intermediate languages rather than changing the machine model.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

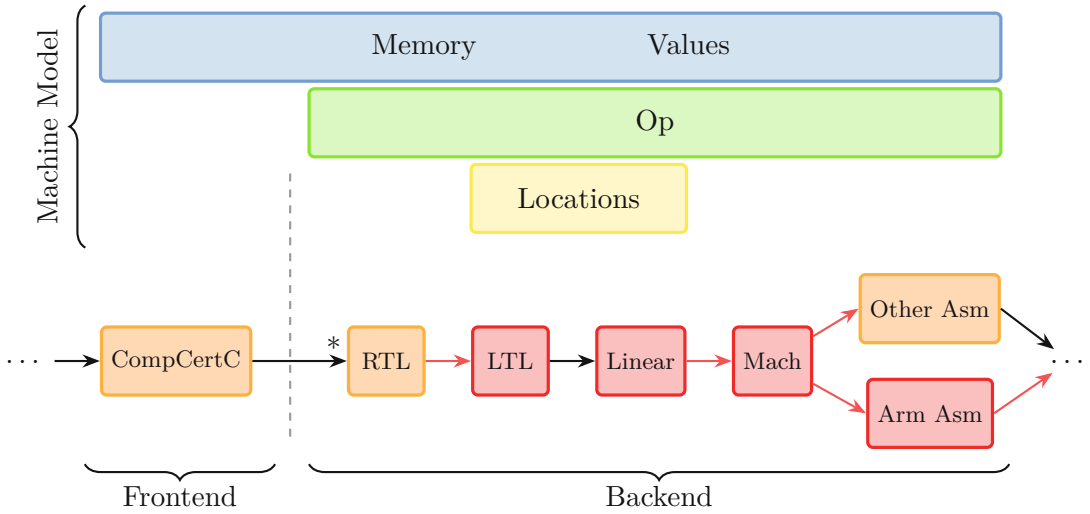


Figure 3.6: Scope of our contributions

We use the type `rpair mreg`, a simple container of either one or two registers, to enable intermediate languages to operate on register pairs or single registers. Alignment constraints, or other restrictions on pairs, are not enforced but rather checked when translating into the respective target's assembly language. Syntactically, the change is minimal and below we give an excerpt of the LTL `instruction` type, highlighting where pairs have replaced the use of single registers. Most occurrences of `mreg` are replaced by `rpair mreg`, but pairs are not supported as arguments of all instructions. The arguments to `Lload`, used to compute the address, are only integer registers and we therefore omit providing support for pairs.

```
Inductive instruction: Type :=
  | Lop (op: operation) (r⃗: list (rpair mreg)) (r: rpair mreg)
  | Lload (chunk: memory_chunk) (addr: addressing) (r⃗: list mreg) (r: rpair mreg)
  | Lcall (sg: signature) (ros: mreg + ident)
  | Lbranch (s: node)
```

Figure 3.7: Selected LTL instructions with pairs

Semantically, we need to support retrieving and storing pairs of registers in an LTL location map. We do not change the location map but rather load the two halves of a pair individually and append the two halves at the bit-level. Storing a value splits it into two halves and then stores them in the individual registers. Establishing the correctness of the repeated splitting and combining of values is done during the proof of semantic preservation between RTL and LTL.

The later translations do not require such an argument since they all handle pairs and perform the same splitting and combining. This is also why the translation between

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

LTL and Linear required no adaptation. The translation from Linear to Mach required adaptations only for the newly inserted code dealing with callee-save registers.

3.3.2 Changes to Assembly Generation & Assembly Semantics

In the Arm semantics, we now model the single-precision registers `S0-S31` instead of the double-precision registers. Instructions in the assembly semantics are typed, only accepting either floating-point or integer registers. The correct use of registers is enforced by the register allocator, but for convenience also rechecked at assembly generation. We make this more fine-grained, restricting the double-precision instructions to pairs and single-precision instructions to single registers.

Modeling all registers enables us to implement the calling conventions correctly. Previously, since only double-precision registers were used, the semantics could not reflect the correct handling of single-precision registers. We can omit any unverified additional code enforcing calling conventions.

The semantics of all intermediate languages that do not use pairs stay unchanged. Assembly generation checks the correct use of registers and we can simply add an additional restriction excluding the use of pairs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Implementation

In this chapter, we highlight some implementation details and discuss how our design choices affected the development. The aim is to present insights into specific parts of the work rather than detailing the entire development accessible online[1]. We start with detailing how we resolve pairs by splitting values in Section 4.1 and then discuss two translation passes in the following sections. We conclude by giving insights into the register allocator in Section 4.4.

4.1 Splitting Values

As outlined in Section 3.2 our contributions affect only the semantics of intermediate languages and the machine model remains as is. To achieve this, we resolve pairs at the level of intermediate languages by splitting and combining values at the bit-level. When a pair is used as an argument of an operation, the two values stored in the individual halves are loaded, combined and then passed as arguments. To store the result of an operation in a pair, the value is split and the halves are stored in the two registers making up the pair. This splitting and combining of values poses four main challenges.

1. Splitting and combining is only well-defined for some values. The bit representation of pointers is not available and therefore splitting is not possible. Moreover, we do not want to give semantics to arbitrary combinations of values, e.g., combining a single-precision float and a long integer.

2. Combining two values requires information about the expected value. We need to be able to reconstruct the correct value, given the two halves. This boils down to picking the right constructor.

[1] `https://github.com/AlexLoitzl/CompCertp`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3. Type information is not sufficient for combining values. Depending on the architecture, pointers have either type `Tlong` or `Tint`, but pointer values cannot be split in two. Moreover, any value has type `Tany64` further complicating the matter.

4. Combining and splitting can destroy values. If we take two arbitrary values, combine them and split them apart again we have no guarantee that we get back the original values.

We define the three operations `combine`, `loword`, and `hiword` that are only defined on `Vdouble` and `Vlong`, and for all others return `Vundef`. The definitions of the first two are given below. The functions `longofwords` and `doubleofsingles` simply append the two binary representations of the arguments together, forming the combined value. The functions `lowordoflong` and `lowordofdouble` split the binary representation in two, returning the lower half as a `Vint` or `Vsingle`.

```
Definition combine (v1 v2: val) :=
  match v1, v2 with
  | Vint _, Vint _ =>
      longofwords v1 v2
  | Vsingle _, Vsingle _ =>
      doubleofsingles v1 v2
  | _, _ => Vundef
  end.
```

```
Definition loword (v: val) :=
  match v with
  | Vlong _ => lowordoflong v
  | Vdouble _ => lowordofdouble v
  | _ => Vundef
  end.
```

Figure 4.1: Functions to split and combine values

Our definitions address the second and third challenges mentioned above. Combining and splitting is only defined on `Vfloat` and `Vlong` and we use the values `Vsingle` and `Vint` to represent their halves, respectively. Representing one half of a `Vdouble` as a `Vsingle`, rather than more naturally as a `Vint`, allows us to correctly recombine the values without requiring any additional typing information.

The remaining two challenges are not directly addressed by our definitions but rather handled during the proofs. Below we show two interesting properties related to our definitions.

Lemma `combine_split_eq` establishes that we can split and recombine a value of type `Tdouble`. We can also prove a similar lemma for `Tlong`, but require an additional hypothesis excluding pointers of type `Tlong`. In the proof of allocation validation, showcased in Section 4.2, we rely on these two lemmas to prove correct splitting and combining of values. This is only possible because of the data-flow analysis connecting the halves to their corresponding value in the RTL code.

The converse, namely first combining two values and then splitting them does not enjoy the same property. Lemma `hiword_combine_eq` is not provable and corresponds to the fourth challenge listed above. Even if we can ensure that both values `v1` and `v2` have

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

type `Tsingle`, it might be the case that `v2` is `Vundef`. Hence, combining and splitting the two might not retain `v1`.

```
Lemma combine_split_eq:
  forall v,
  Val.has_type v Tdouble ->
  v = combine (hiword v) (loword v).
```

```
Lemma hiword_combine_eq:
  forall v1 v2,
  Val.has_type v1 Tsingle
  /\ Val.has_type v2 Tsingle ->
  v1 = hiword (combine v1 v2).
```

Figure 4.2: Lemmas to reason about splitting and combining values

4.2 Allocation Validation

Proving the correctness of allocation validation was one of the main challenges of introducing pairs to CompCert. We need to verify that the result of the new register allocator, now using pairs, preserves the semantics of the RTL code. We adapt the algorithm reviewed in Section 2.5.1 to support reasoning about the repeated splitting and recombining of pairs. We retain the two phases, first performing structural checks and then a data-flow analysis. Below we discuss the changes to the algorithm highlighting two interesting cases of the transfer function.

4.2.1 Structural Checks

In addition to the existing checks, we now enforce the correct use of register pairs. We check that the two registers in a pair are distinct and that two pairs involved in a move do not overlap. If these two properties hold, and, as outlined above, we can split a value correctly, we can view a move between two pairs, as two moves between the individual halves. We do not enforce any architectural constraints like alignment of pairs as the correct use of registers is enforced during assembly generation.

4.2.2 Data-Flow Analysis

The data-flow analysis requires more significant changes. We need to relate the values flowing through the RTL and LTL programs, where in the latter we repeatedly split and recombine values in register pairs. The original algorithm uses equation sets to relate values in temporaries and locations and already offers some support for splitting 64-bit integers. The main difference in CompCert^p is that we can update the two registers of a pair simultaneously with a single LTL instruction. This also needs to be reflected in the equation sets and we considered two options.

1. **Modeling pairs in equations**: Rather than equations relating RTL temporaries with locations, they relate temporaries with location pairs (`rpair loc`), explicitly connecting the two halves of a pair.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2. **Keeping the equation sets as they are**: Equations already come in three kinds to enable the relation of a location containing one half of a split 64-bit integer with the temporary containing the entire value. Extending this to floating-point values is easy, but the two halves of a pair are treated as two unrelated registers.

The first option is attractive as it keeps track of pairs and allows performing a single update on the equation set to reflect an assignment to a pair. The downside is that the location pairs of two equations can alias and updates to the equation set need to check all other equations for aliasing. This complicates an efficient implementation of equation sets preventing fast implementation of routine operations like updates and compatibility checks.

We therefore implemented the second option, extending the use of equation kinds to floating-point values. Keeping the equations as they are allows for a straightforward adaptation of many cases of the transfer function, including that of arithmetic operations. The main difficulty is reasoning about the inserted moves which now requires strong assertions to relate the two halves of a pair. Below we first define some new operations on equation sets and then showcase the updated transfer function.

We write $x \approx p$ to relate a temporary x with a location pair p (`rpair loc`).

$$x \approx p \stackrel{def}{=} \begin{cases} \lfloor \{x =_F m\} \rfloor & , \text{ if } p = \texttt{One } m \\ \lfloor \{x =_L lo, x =_H hi\} \rfloor & , \text{ if } p = (\texttt{Two } hi\ lo) \text{ and } \mathit{splittable}(x) \\ \top & , \text{ otherwise} \end{cases}$$

The predicate $\mathit{splittable}(x)$ checks that x can be split and recombined without destroying the value stored in x. This amounts to checking that x has type `Tdouble` or that it has type `Tlong` and pointers have type `Tint`.

We define the intersection between a location l and equation set E as all equations that relate l to some temporary.

$$l \cap E \stackrel{def}{=} \{(x =_\kappa l') \mid (x =_\kappa l') \in E \wedge l = l'\}$$

We overload $\perp$ to check compatibility of a location l with an equation set E for kinds $K \subseteq \{F, H, L\}$. A location l is compatible with E for kinds $k \in K$ if all equations either contain l and are of kind $k \in K$ or the location does not overlap with l.

$$l \perp_K E \stackrel{def}{=} \forall (x' =_\kappa l') \in E : \ (x' = x \wedge l' = l \wedge \kappa \in K) \vee (x' \neq x \wedge l' \perp l).$$

Below we detail the adapted transfer function. The case for the arithmetic operations is similar to before, now adding and removing two equations in case of register pairs. Since operations are typed, we can check that pairs are only used for arguments or results of type `Tlong` and `Tdouble`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The case for inserted moves now requires multiple checks to ensure correct splitting and combining of values. As outlined in Section 4.1, combining two values and then splitting them apart may not preserve the two values. To reason about the result of the move we check that the two registers of a pair are related to the upper or lower halves of the same RTL temporaries.

```
transfer_instr instr bblock E :=
  match instr, bblock with
  | Iop _ x̄ x _, Lop _ l̄ l _ =>
      if (x,l) ⊥ E then E \ {x ≈ l} ∪ {x̄ ≈ l̄} else ⊤
  | _, (Lop mov (Two dst_hi dst_lo) (Two src_hi src_lo)) :: bblock =>
      if dst_hi ∩ E = dst_lo ∩ E /\ dst_lo ⊥_{L,H} E /\ dst_hi ⊥_{L,H} E
      then let E' := transfer_instr instr bblock E[dst_lo ← src_lo] in
           transfer_instr instr block E'[dst_hi ← src_hi]
      else ⊤
  end.
```

Figure 4.3: Parts of the new transfer function

Proving Soundness

The soundness proof proceeds the same as for the original algorithm. Below we state two lemmas that allow us to reason about the two cases of the transfer function given above.

Lemma 4.2.1 lets us reason about the simultaneous assignment of a value v to a temporary x and a location pair (hi, lo). This is similar to the original case but amounts to checking the constraints for each half individually.

Lemma 4.2.1 (Simultaneous Pair Assignment)
If $rs, ls \models E \setminus \{(x =_H hi), (x =_L lo)\}$, $(x, lo) \perp E$, *and* $(x, hi) \perp E$,
then $rs[x \leftarrow v], ls[hi \leftarrow \texttt{hiword}\ v][lo \leftarrow \texttt{loword}\ v] \models E$.

Lemma 4.2.2 corresponds to the second case of the updated transfer function. It highlights the downside of the chosen representation of equation sets. The equation set only allows updates of each half of the pair individually, while on the level of the location map, we perform a single update. We write $ls[(dst_{hi}, dst_{lo}) \leftarrow ls((src_{hi}, src_{lo}))]$ to signify that the update is simultaneous. In the background, this amounts to, loading each half src_{hi}, and src_{lo} individually, combining the value, splitting it again and assigning to dst_{hi}, and dst_{lo}. The additional hypotheses of the lemma serve to ensure that this is equivalent to the two updates of the equation set.

Lemma 4.2.2 (Inserted Pair Move)
If $E[dst_{hi} \leftarrow src_{hi}][dst_{lo} \leftarrow src_{lo}] \neq \top$ *and* $rs, ls \models E[dst_{hi} \leftarrow src_{hi}][dst_{lo} \leftarrow src_{lo}]$,
and $dst_{hi} \cap E = dst_{lo} \cap E$, *and* $dst_{lo} \perp_{\{L,H\}} E$, *and* $dst_{hi} \perp_{\{L,H\}} E$,
and $dst_{lo} \neq dst_{hi}$, *and* $dst_{hi} \neq src_{lo}$,
then $rs, ls[(dst_{hi}, dst_{lo}) \leftarrow ls((src_{hi}, src_{lo}))] \models E$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.3 Preservation of callee-save registers

Up to the Mach intermediate language the automatic preservation of callee-save registers was built into the semantics of a function call. This is resolved during Stacking, the translation pass from Linear to Mach. It determines the used callee-save registers of a function and inserts instructions into the function prologue and epilogue that ensures their preservation across a function call. The new instructions do not have any corresponding ones in the previous intermediate languages. This sets the Stacking pass apart from the other translation passes affected by our changes.

CompCert^p exposes a more fine-grained view of the registers, giving us three possibilities to handle the preservation of callee-save registers.

1. **Only saving pairs**: This approach saves the entire pair if any of its subregisters is used by the function. The downside of this approach is the slight increase of the function's stack frame as we are also saving registers that are not used by the function.

2. **Saving individual registers**: Saving all registers individually prevents wasting any space in the stack frame. It comes at the cost of a greater code size, as we need two instructions to preserve a pair if both halves are used.

3. **Saving pairs only if both registers are used**: This approach is the best of both worlds. If only one register is used we only store one half, not wasting any stack space. If both registers of a pair are used by a function, we only use one instruction saving both halves.

Initially, while proving the soundness of CompCert^p we first followed the second approach. CompCert has no notion of pairs and while it stores individual registers, its strategy for Arm corresponds to the first approach. Since only double-precision registers are used, even if they only hold a single-precision value, the entire register is stored on the stack. In CompCert^p, we switched to modeling single-precision registers and by following the same strategy as CompCert we implemented the second approach.

Implementing the third approach is difficult since no information about the values stored in the registers is available. The contents of the registers stem from the calling function and we cannot even hope to compute a precise type at compile time. This is the main difference to the proof of the allocation validation detailed above. There, two halves of a value were always related to the entire value in the RTL program.

The problem we encounter corresponds to the second, unprovable lemma in Section 4.1. If we store a pair of registers containing unrelated values, we cannot prove that we can retrieve the two original values. For the sake of producing more efficient code, we implement approach three for CompCert^p, treating the action of saving and restoring callee-save registers differently than a normal store or load.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

We introduce the two new instructions `Msavecallee` and `Mrestorecallee` to the Mach intermediate language. The two instructions store to and load from the stack. Unlike other instructions, they do not combine and split values but rather treat the two halves individually. Below we showcase the semantics of `Msavecallee` for a register pair. Conceptually easy, it also highlights the special care required if we do not perform the splitting and combining at the level of values. The instruction needs to be endian aware, storing the correct half at the lower address. In addition, we need to explicitly compute the offset to store the second register.

```
Definition save_callee_rpair m sp ofs p rs :=
  match p with
  | Two r1 r2 => let rhi:= if big_endian then r2 else r1 in
                 let rlo := if big_endian then r1 else r2 in
                 let κ := type_of rlo in
                 let κ' := type_of rhi in
                 if m[(sp, ofs) ←κ rs(rlo)] = ⌊m'⌋
                 then m'[(sp, ofs + |κ|) ←κ' rs(rhi)]
                 else ⊤
  end.
```

Figure 4.4: Pseudocode of `save_callee_rpair` for pairs

For the correctness proof we need to show that we can save the register contents onto the stack using `Msavecallee` instructions and later reload the correct value using `Mrestorecallee` instructions. We define a separating conjunction `contains_rpair` depicted below. It is quite similar to `save_callee_rpair` and has a matching `contains` predicate for each store in `save_callee_rpair`. Intuitively, the `contains` predicate states that the memory `m` at the given block `sp` and offset `ofs` contains a value `v` of a certain type κ. The `**` is the star operator from separation logic ensuring disjoint memory regions.

```
Definition contains_rpair sp ofs p rs :=
  match p with
  | Two r1 r2 => let rhi := if Archi.big_endian then r2 else r1 in
                 let rlo := if Archi.big_endian then r1 else r2 in
                 let κ := type_of rlo in
                 let κ' := type_of rhi in
                 contains κ sp pos (rs rlo)
                 ** contains κ' sp (pos + |κ|) (rs rhi)
  end.
```

Figure 4.5: Simplified separating conjunction

Equipped with the `contains_rpair` predicate above, we can formulate a lemma capturing the use of `save_callee_rpair` to save a register pair to the stack. As expected, the lemma

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

allows us to establish that after saving a register to the stack using `callee_save_rpair`, the memory satisfies the `contains_rpair` predicate, stating that the memory contains the two values stored in the register pair. The first two assumptions ensure that the registers have the same size and that the size of the pair divides the offset being stored to. This is a requirement to satisfy alignment constraints of the stack. The third assumption states that the memory has a disjoint range from `ofs` to `(ofs + size_of p)` in block `sp` with write access.

```
Lemma contains_rpair_save_callee_rpair:
  forall (spec: mreg -> val -> Prop) spec1 m sp ofs P ls p rs,
    wf_pair p ->
    size_of p | ofs ->
    m |= range b ofs (ofs + size_of p) ** P ->
    exists m',
      save_callee_rpair m sp ofs p rs = Some m'
    /\ m' |= contains_rpair sp ofs p rs ** P.
```

Figure 4.6: Selected LTL instructions with pairs

The proof establishing the correct preservation of the callee-save registers uses the properties and functions showcased above. We can state and prove a similar lemma to `contains_rpair_save_callee_rpair` that allows us to correctly reload the values from the stack given it satisfies the separating conjunction.

The changes to the Mach semantics also need to be reflected in the Arm assembly semantics. We adapt the semantics of the `Pfldd_a` and `Pfstd_a` instructions, corresponding to the Arm instructions `vldr` and `vstr`. The instructions have been introduced to CompCert to handle the loading and storing of callee-save registers. We adapt them to reflect the semantics of `Msavecallee` and `Mrestorecallee`, similarly storing the individual registers and explicitly computing the offset. This is defined in accordance with the Arm semantics [Armb]. The architectures that do not support register pairs do not need to be adapted.

4.4 Register Allocation

CompCert's implementation of IRC closely follows the pseudocode given in [GA96], but allocates both integer and floating-point registers at the same time. Our allocator, similar to GRA, builds on top of the existing implementation. Each architecture needs to export an interface to the allocator. The changes to the allocator itself are then small and mostly consist of replacing constants and hard-coded assumptions with architecture-dependent functionality.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.4.1 Architecture Interface

We equip each architecture with an interface detailing its register classes and how they interact. Since the registers available to the allocator may differ from those modeled in CompCert^p's semantics, each architecture also provides functions to map back and forth between the two.

In the case of Arm, the allocator adopts the view of the two classes S and D of single-precision and double-precision floating-point registers. The assembly semantics on the other hand, only use the registers in S and the double-precision registers are modeled explicitly as a pair of registers. For convenience, we extend the type `mreg` with the registers from D, but do not model any aliasing. If the LTL code output by the allocator would use any registers from D, the compiler would abort.

Below we give a pseudo-code excerpt from the interface exposed by Arm. On the left, we see how classes and their relationship are modeled. On the right, the function `expand_mreg` implements the mapping between the registers used in the allocator to the type `rpair mreg` used by LTL. We also define the concrete aliasing between any two specific registers in `regs_alias`.

```
let classes = [I; S; D]

let class_of_reg r =
  match r with
  | R0 | ... | R12 -> I
  | F0 | ... | F31 -> S
  | D0 | ... | D15 -> D

let classes_alias c1 c2 =
  match c1, c2 with
  | S, D | D, S -> true
  | _, _ -> (c1 = c2)

let worst class1 class2 =
  match class1, class2 with
  | I, I | S, S | D, D -> 1
  | S, D -> 2
  | D, S -> 1
  | _, _ -> assert false
```

```
let expand_mreg m =
  match m with
  | D0 -> Two (F1, F0)
  | D1 -> Two (F3, F2)
            ⋮
  | D15 -> Two (F31, F30)
  | mr -> One mr

let regs_alias r1 r2 =
  let alias f1 f2 =
    match f1, f2 with
    | F0, D0 | F1, D0 -> true
    | F2, D1 | F3, D1 -> true
      ⋮
    | F30, D15 | F31, D15 -> true
    | _, _ -> false
    in
  r1 = r2 || alias r1 r2 || alias r2 r1
```

Figure 4.7: Architecture interface of Arm

The interface exported by the other architectures is quite simple and reflects the assumptions the allocator previously made about every architecture. There are two disjoint classes of floating-point and integer registers that do not alias. Every neighbor increases the degree of a node by one. The registers used by the allocator are the same as those

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

modeled in the assembly semantics, hence a trivial mapping into the type `rpair mreg` is defined. Since distinct registers do not alias, we check for equality.

```
let classes = [I; F]

let class_of_reg r =
  if is_float_reg r then F else I

let classes_alias c1 c2 = (c1 = c2)
```

```
let worst class1 class2 =
  match class1, class2 with
  | I, I | F, F -> 1
  | _, _ -> assert false

let expand_mreg m = One m

let regs_alias r1 r2 = (r1 = r2)
```

Figure 4.8: Generic architecture interface

4.4.2 Changes to the Allocator

The changes to the allocator are simple and consist of replacing hard-coded assumptions about register classes with the architecture interfaces defined above. In many places, we change equality tests between registers with alias tests and degree computations use *worst*.

While illustrating the interference graph as a weighted directed graph is convenient, we do not change the internal representation of the graph. IRC caches the degree and the neighbors of each node and incrementally updates the two whenever necessary. For example, during the construction of the graph, when an edge is added, the degree of each node is increased and the interfering nodes are added to each other's adjacency lists. Instead of incrementing the degree by one, we use *worst* to compute the change. While in [SRH04], Smith et al. report on reusing IRC's coalescing criterion for GRA, we found that introducing register classes can lead to faulty coalescing of nodes. Below we explore the problem in more detail.

Coalescing & Constrained Moves

In [GA96], the notion of a *constrained move* is introduced. A move is constrained if it cannot be coalesced or frozen. In IRC, constrained moves occur if two nodes that are move-related also interfere, as depicted on the left in Figure 4.9. When generalizing the allocator to support aliasing machine registers, constrained moves can take more complex shapes.

In the classic setting without aliasing registers, one can include the K machine registers in the graph as a K clique to prevent any coalescing[CACC+81]. When introducing register classes, this view cannot be adopted as two aliasing machine registers cannot be live at the same time. Hence, the absence of an interference edge between two machine registers signals that they do not have overlapping lifetimes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Figure 4.9: Constrained moves

The allocator needs to ensure that two aliasing machine registers are never connected by an interference edge. Hence, a move between two nodes u and v is constrained if either, u and v interfere or if the resulting node aliases with one of its neighbors. The second case is depicted in Figure 4.9 on the right. The machine registers `Do` and `So` alias. If we coalesce the move, we introduce an interference edge between the two and create an overlapping lifetime.

Parallel Moves

One component of the register allocator of CompCert that we have not mentioned so far is the algorithm to perform parallel moves. A parallel move is an assignment $(x_1, \ldots, x_n) \coloneqq (y_1, \ldots, y_n)$, such that after the assignment each x_i contains the value stored in y_i. A simple serialization $x_1 \coloneqq y_1; \ldots; x_n \coloneqq y_n$ can lead to a different outcome than the parallel assignment, if there are some i, j, such that $x_i = y_j$ and $i \neq j$. The simplest such case is the parallel move $(x_1, x_2) \coloneqq (x_2, x_1)$.

Parallel moves are not part of any of CompCert's intermediate languages and are only used to honor calling conventions. The RTL code uses temporaries to pass arguments while the LTL code uses fixed machine registers prescribed by the calling conventions. After an execution of IRC CompCert inserts a serialized parallel move between the registers assigned to the RTL temporaries and those used in the LTL code.

The algorithm used by CompCert relies on a fresh temporary to serialize the parallel move in linear time. It views the parallel move as a directed graph in which the registers are the node and a node u is connected to v if there is a move from v to u. These graphs are of a tree-like shape, but may also contain disjoint cycles. A detailed description of the graphs and the concrete algorithm used is given in [RSL08].

Introducing aliasing between registers destroys the property of disjoint cycles. We therefore instrument IRC to ensure that the destination and sources of a parallel move do not partially overlap. We achieve this by introducing additional interference edges when constructing the interference graph. Hence, for any two registers x and y in a parallel move, either $x = y$ or $x \perp y$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Evaluation

The two big contributions of CompCert^p are the new register allocator and the adapted backend now supporting register pairs. In total, the git statistics report around 7000 (5000+/2000-) changed lines compared to CompCert. Using `coqwc` and `ocamlwc` we give a detailed breakdown of the additional lines of code.

	backend	Arm	other architectures (Avg.)
Coq Implementation	613	34	53.6
Coq Proofs	795	261	51.2
Ocaml Code	83	119	5

Table 5.1: Additional lines of codes of CompCert^p

Note that the numbers reported in the table above only capture the overhead of CompCert^p. We successfully managed to keep the additions to the unaffected architectures minimal. The bigger increase in proofs for the Arm architecture is due to the handling of the two new instructions `Msavecallee` and `Mrestorecallee` we introduced to the Mach intermediate language. In the backend, proofs and implementation grew at the same rates, which is similar to the existing development.

The development of CompCert^p was carried out on CompCert (Release 23.10) distributed by AbsInt [1] and all tests and the statistics above compare CompCert^p against this release. The contributions published online[2] have been ported to the public release.

[1] `https://www.absint.com`
[2] `https://github.com/AlexLoitzl/CompCertp`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.1 Benchmarks

In the following, we compare compile time, code size of the generated object files, and statistics of the register allocator. We use three benchmark suites described below.

1. SPEC CPU 2000[3]: The SPEC CPU benchmarks are a popular benchmark suite used by industry and academia. We use the benchmarks *175.vpr, 176.gcc, 186.crafty, 254.gap, 255.vortex,* and *300.twolf* from the CINT2000 suite and *177.mesa, 179.equake,* and *188.ammp* from the CFP2000 suite.

   Since CompCert only supports C, we were limited in the choice of benchmarks, especially for the floating-point suite which only includes 4 C benchmarks. We did not use two benchmarks with very short compile times (*256.bzip2, 179.art*) and slightly modified the used benchmarks to comply with C99. The benchmarks are therefore so-called non-reportable runs according to the SPEC rules.

2. absint: This benchmark suite was provided by AbsInt and contains over 5000 test cases comprised of industry code and commercial compiler benchmarks.

3. fuzz: This benchmark suite is split into *fuzz1, fuzz2*, and *fuzz3* each consisting of 100 files. It includes generated test cases that contain increasingly complex floating-point expressions and function calls using floating-point parameters. We use it both as a stress test of our register allocator and to highlight the best-case improvement our contributions can provide. We make the test cases available online together with CompCert^p.

We ran all tests on a notebook with an AMD Ryzen 7 Pro 5850U CPU (8 Cores, 16 hardware threads, 1.9/4.4GHz clock) and 32GB of DDR4 RAM running Debian *trixie* (kernel 6.6.15). In an attempt to decrease system noise, we ran all tests in recovery mode, set up a shielded set of cores and set the CPU scaling governor to *performance*.

In the following, we compare the compile time and generated code size of CompCert and CompCert^p. We run all tests on the Risc-V target as a representative for the architectures that do not support register pairs. For Arm, we run all tests on the two supported Arm ABIs. The hard float ABI corresponds to the calling conventions presented in Section 2.2. The soft float ABI uses the standard calling conventions to pass arguments in the registers `R0-R3` but otherwise uses the floating-point registers. This can be used to link with a library that uses emulated software floating-point arithmetic. In this chapter we highlight the biggest differences and include the remaining measurements in the appendix.

5.1.1 Compile Time

Figure 5.1 shows the compile times of CompCert^p in relation to CompCert. The measurements for the SPEC benchmarks were repeated 10 times, the others 5 times. We

[3] `https://www.spec.org/cpu2000`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

show the standard deviation with red error bars. We used `perf`[4] to get timings of the different phases during compilation, giving us an insight into the various tests and how hard their allocation problems are. In most cases register allocation and the allocation validation increase slightly.

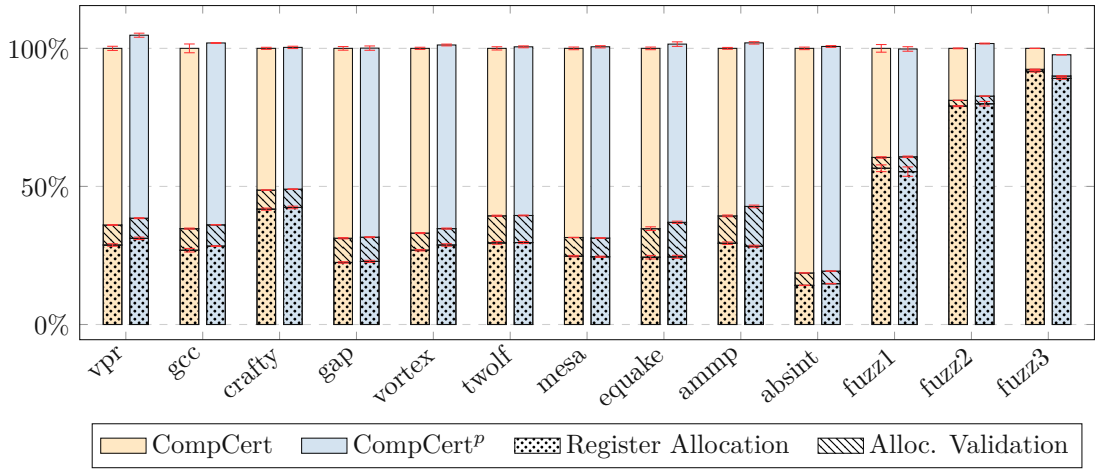


Figure 5.1: Compile times for the Arm target (Hard float)

In Table 5.2 we capture the average change in execution time across all benchmark suites for all architectures. We see a slight increase in total compilation time, with the Arm soft float target being an outlier. This is due to the single big improvement on the fuzz3 benchmark, which might be a skewed measurement. Allocation validation takes considerably longer for the Arm targets due to the two benchmarks *ammp* and *equake* in which the time increases by 45.54% and 21.02% respectively. Since allocation validation is usually short, this does not contribute a big increase to the total compile time.

	arm_hard	arm_soft	riscv
Register Allocation	+1.63%	-0.04%	+4.00%
Allocation Validation	+11.61%	+11.94%	+1.17%
Remaining Translations	+0.24%	-0.12%	+0.59%
Total compile time	+0.99%	-0.29%	+1.91%

Table 5.2: Compile times split into phases

5.1.2 Code Size

Most examples did not change significantly in code size. Since the allocation algorithm uses heuristics to color the graph, changes to the layout of the graph already lead to

[4] `https://perf.wiki.kernel.org`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

small changes in the output code. We therefore see small changes in the code size when compiling to Arm. For the targets that do not use register pairs, our allocator outputs identical code. Below we list the tests for which we recorded the biggest changes.

	vpr	mesa	fuzz1	fuzz2	fuzz3
arm_hard	-0.95%	-1.97%	-8.38%	-8.78%	-9.60%
arm_soft	-0.16%	-0.65%	-0.36%	+0.22%	+0.39%

Table 5.3: Changes in generated code size between CompCert and CompCert^p

We see the biggest improvements in our generated test cases for the Arm hard float target. Omitting the unverified pass that inserts moves around function calls gives us a significant decrease in code size. For the Arm soft float target we record less significant changes and for some of the generated test cases even see a small increase in code size.

5.1.3 Allocation

For the cases in which we see more significant changes, we list more detailed statistics of the allocator in Tables 5.4 and 5.5. We capture the moves remaining in the program after register allocation and the moves inserted by the unverified pass for the hard float ABI. We record reloads and spills and the average number of iterations per function. For the fuzzed suites we take the average of all contained test cases.

	Remaining		Inserted	Reloads		Spills		Iterations	
	C	C^p	C	C	C^p	C	C^p	C	C^p
vpr	4557	4557	253	275	275	298	297	2.67	2.67
mesa	13414	13420	1341	1401	1276	2265	2133	1.4	1.32
fuzz1	119	118	84	17	17	17	15	1.16	1.14
fuzz2	404	404	307	115	115	74	64	1.26	1.24
fuzz3	1515	1514	1184	456	461	267	226	1.43	1.39

Table 5.4: Selected register allocation statistics for Arm (Hard float)

For both Arm targets, we see a decrease of up to 10% in spills and reloads and a decrease in iterations, which also explains the less significant changes in compile time for the Arm targets compared to the Risc-V target. For the hard float Arm target, we also see the impact of the inserted move instructions. For the two SPEC benchmarks, CompCert increases the number of moves by 5% and 10%. For our generated test cases with many function calls the inserted moves contribute considerably to code size.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

	Remaining		Reloads		Spills		Iterations	
	C	C^p	C	C^p	C	C^p	C	C^p
vpr	4557	4557	275	275	298	297	2.67	2.67
mesa	13415	13420	1398	1276	2260	2133	1.4	1.32
fuzz1	119	119	17	17	17	15	1.16	1.14
fuzz2	404	404	115	115	74	67	1.26	1.23
fuzz3	1515	1514	456	460	268	231	1.44	1.39

Table 5.5: Selected register allocation statistics for Arm (Soft float)

5.2 Related Work

5.2.1 Register pairs in CompCert

In [Bar18], Barany reports on an effort to introduce register pairs to CompCert. Instead of splitting values to store individual halves, the register and stack are modeled as a block of bytes. This requires typing information in the backend to translate between values and sequences of bytes stored in the register set. This model of the register file should support register pairs where one half can be accessed with an offset into the pair. Support for pairs was never added and the changes have not been included in CompCert.

5.2.2 Aliasing Registers in Graph-Coloring Register Allocation

Early approaches to support register pairs [BCT92] and addressable aligned consecutive registers [Nic90] focused on modifying the interference graph to represent the additional constraints. Unlike the approach we took, they are constrained to certain kinds of aliasing relationships.

An early predecessor of GRA, presented in [SH00], puts weights on the nodes of the interference graph based on the class of the temporary. This approach can model arbitrary aliasing relationships but may be more conservative during *Simplify*. GRA focuses on optimizing the degree computation of a node by constructing a class tree capturing the aliasing relationships between register classes. Each node caches the contribution to its degree per class and performs recursive updates of the cached values in its class tree.

CompCert only uses GPRs (type `mreg`) during allocation and the class trees for all supported architectures only consist of a single root node. We therefore omit the unnecessary caching and recursive computation of degree changes, giving us a small performance benefit and allowing a more natural interpretation of the interference graph. This simplified version of GRA corresponds to the ideas presented in [RN03; RN02]. They implement their strategy only on a simplified version of Chaitin's algorithm without coalescing.

None of the above works reports on parallel moves or necessary changes to coalescing. Since we have precolored temporaries to enforce calling conventions, aggressive coalescing leads to errors in the allocation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Conclusion

One of the central efforts of safety-critical software development lies in the various verification means to ensure correctness. The CompCert formally verified compiler reduces the required verification steps via a machine-checkable proof of semantic preservation. The proof ensures the correct translation from a C-like intermediate language to the modeled semantics of the target's assembly language.

The aim of this thesis has been to adapt CompCert to support register pairs, a common hardware feature in which two registers are used to hold a single value. To this end, we implemented CompCert^p, a simple extension of CompCert that can be easily integrated into the main development.

CompCert^p's adapted backend supports register pairs such that the semantics of architectures with no support are not affected. We revise the Arm semantics of CompCert to include register pairs, allowing us to correctly implement the calling conventions for floating-point arguments in the proven part of CompCert^p. Unlike CompCert, we do not rely on unverified code to comply with the floating-point calling conventions of Arm. CompCert^p's Arm semantics model those of the architecture specification more precisely than CompCert, therefore increasing the trust in the correctness proof.

We perform extensive tests on well-known benchmarks and generated test cases showing that CompCert^p either generates smaller or similar code with a slightly increased compile time. On examples with many function calls and floating-point arithmetic, we achieve a decrease of up to 10% in code size.

6.1 Future Work

Currently, CompCert's only supported target with register pairs is Arm, while support for the TriCore architecture is planned for a future release. Similar to Arm, it also uses

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

register pairs to support double-precision floating-point arithmetic. The next step is to model register pairs in the TriCore semantics and make use of the new backend.

For register allocation, there are two open directions to investigate further. Currently, we handle parallel moves by instrumenting the register allocator to prevent any overlap between registers participating in the move. An algorithm that can handle parallel moves between aliasing registers could help to decrease the register pressure around parallel moves during allocation.

CompCert^p reuses the spilling heuristics of CompCert and does not take register classes into account. A more sophisticated approach may tune the spilling heuristics for each architecture and their aliasing relationships. Since registers from different classes have different effects on each other, favoring one over the other might improve spilling.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

ABI Application Binary Interface. 2, 3, 25, 44, 46

CFG Control-flow graph. 12, 13

GPR General purpose register. 6, 7, 26, 47

GRA Generalized Register Allocator. 25, 26, 38, 40, 47

IRC Iterated Register Coalescing. 17, 18, 25, 38, 40, 41, 51

TCB Trusted Computing Base. 2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[AAMP+17] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: a verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[App11] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011.

[Arma] Arm Holdings plc. Fye24-q2 results presentation. URL: `https://investors.arm.com/financials/quarterly-annual-results` (visited on 04/14/2024).

[Armb] Arm Limited. Arm architecture reference manual for a-profile architecture. Version J.a. URL: `https://developer.arm.com/documentation/ddi0487` (visited on 04/14/2024).

[Bar18] Gergö Barany. A more precise, more correct stack and register model for CompCert. In *LOLA 2018 - Syntax and Semantics of Low-Level Languages 2018*, Oxford, United Kingdom, July 2018.

[BBFL+12] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2 2012: Embedded Real Time Software and Systems*, Toulouse, France. AAAF, SEE, February 2012.

[BBGH+19] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Lett. Program. Lang. Syst.*, 1(1):3–13, March 1992.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[BDP14] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1), March 2014.

[CACC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, Boston, Massachusetts, USA. Association for Computing Machinery, 1982.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[GHC] GHC Team. The glasgow haskell compiler. Version 9.8.2. URL: `https://www.haskell.org/ghc/` (visited on 04/14/2024).

[GNU] GNU Project. The gnu compiler collection. Version 13.2. URL: `https://gcc.gnu.org` (visited on 04/14/2024).

[Inf] Infineon Technologies AG. Company presentation. URL: `https://www.infineon.com/cms/en/about-infineon/press/general-information/facts-figures` (visited on 04/14/2024).

[JJKD17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[KBWS+18] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, Toulouse, France. 3AF, SEE, SIE, January 2018.

[LABS14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model. In Andrew W. Appel, editor, *Program Logics for Certified Compilers*. Cambridge University Press, April 2014.

[LBDJ+] Xavier Leroy, Sandrine Blazy, Zaynah Dargaye, Jacques-Henri Jourdan, Michael Schmidt, Bernhard Schommer, and Jean-Baptiste Tristan. Compcert, the formally-verified c compiler. Version 3.13. URL: `https://github.com/AbsInt/CompCert` (visited on 04/14/2024).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[Ler09a] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[Ler09b] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler23] Xavier Leroy. The compcert c verified compiler: documentation and user's manual, version 3.13, 2023.

[MB22] David Monniaux and Sylvain Boulmé. The trusted computing base of the compcert verified compiler. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 204–233, Cham. Springer International Publishing, 2022.

[Nic90] Brian R Nickerson. Graph coloring register allocation for processors with multi-register operands. *ACM SIGPLAN Notices*, 25(6):40–52, 1990.

[RL10] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In Rajiv Gupta, editor, *Compiler Construction*, pages 224–243, Berlin, Heidelberg. Springer Berlin Heidelberg, 2010.

[RN02] Johan Runeson and Sven-Olof Nyström. Generalizing Chaitin's algorithm: Graph-coloring register allocation for irregular architectures. Technical report, Uppsala University, 2002.

[RN03] Johan Runeson and Sven-Olof Nyström. Retargetable graph-coloring register allocation for irregular architectures. In Andreas Krall, editor, *Software and Compilers for Embedded Systems*, pages 240–254, Berlin, Heidelberg. Springer Berlin Heidelberg, 2003.

[RSL08] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.

[RTCA11] Radio Technical Comission for Aeronatuics. *Software Considerations in Airborne Systems and Equipment Certification*. Washington, D.C., USA, 2011.

[SCKK+19] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. Compcertm: compcert with c-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[SH00] Michael D. Smith and Glenn Holloway. Graph-Coloring Register Allocation for Irregular Architectures. Technical report, Harvard University, 2000.

[SLKM+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA. Association for Computing Machinery, 2021.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[SLZS16] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, New York, NY, USA. Association for Computing Machinery, 2016.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, 2004.

[TMKF+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA. Association for Computing Machinery, 2011.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Appendix

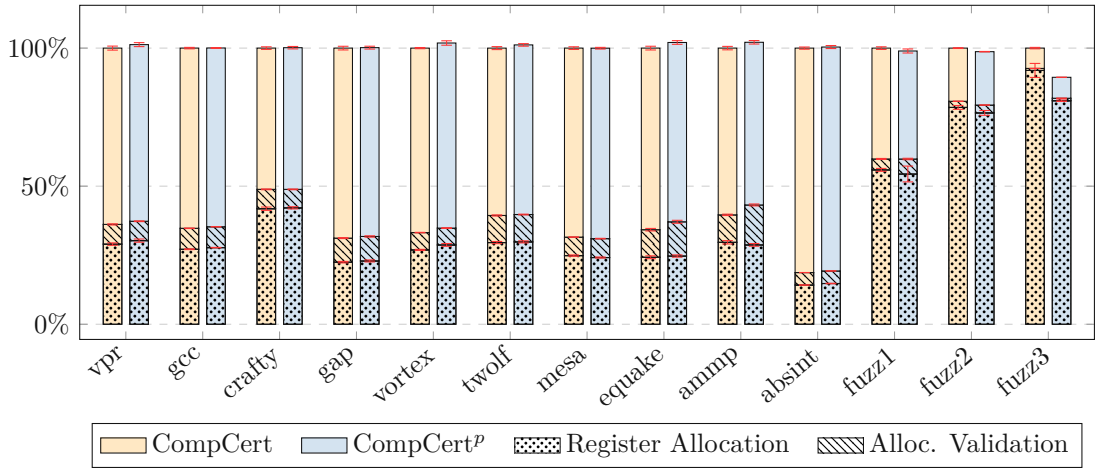


Figure A.1: Compile times for the Arm target (Soft floats)

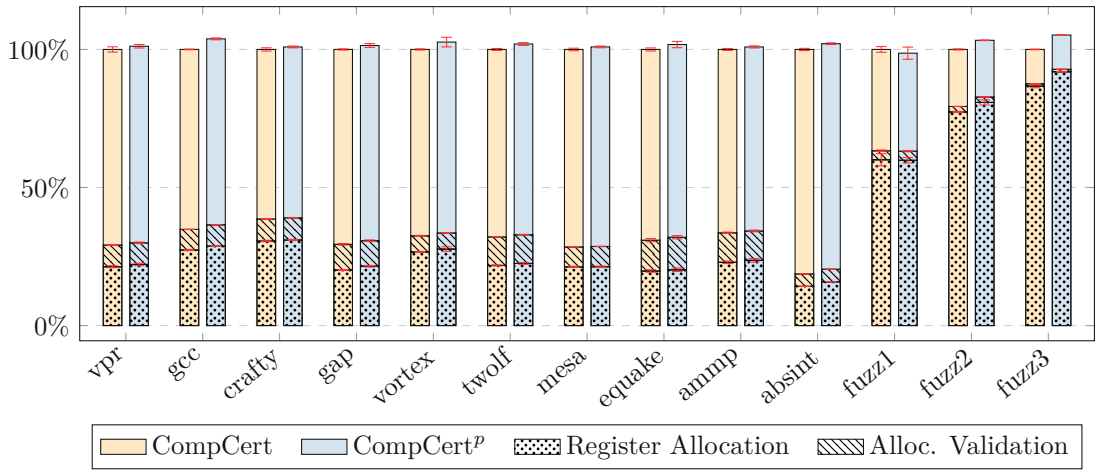


Figure A.2: Compile times for the Risc-V 32bit target

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	Remaining		Inserted	Reloads		Spills		Iterations	
	C	C^p	C	C	C^p	C	C^p	C	C^p
vpr	4557	4557	253	275	275	298	297	2.67	2.67
gcc	52810	52810	0	1125	1125	2006	2006	2.41	2.41
crafty	19501	19501	0	512	512	240	241	1.91	1.91
gap	20145	20145	0	390	390	869	869	2.81	2.81
vortex	19542	19542	0	266	266	324	324	2.82	2.82
twolf	8888	8888	0	342	343	506	506	2.12	2.16
mesa	13414	13420	1341	1401	1276	2265	2133	1.4	1.32
equake	631	631	0	1	1	2	2	1.0	1.0
ammp	3995	3995	0	114	114	333	333	1.82	1.82
fuzz1	119	118	84	17	17	17	15	1.16	1.14
fuzz2	404	404	307	115	115	74	64	1.26	1.24
fuzz3	1515	1514	1184	456	461	267	226	1.43	1.39

Table A.1: Register allocation statistics for Arm (Hard Float)

	Remaining		Reloads		Spills		Iterations	
	C	C^p	C	C^p	C	C^p	C	C^p
vpr	4557	4557	275	275	298	297	2.67	2.67
gcc	52810	52810	1125	1125	2006	2006	2.41	2.41
crafty	19501	19501	512	512	240	241	1.91	1.91
gap	20145	20145	390	390	869	869	2.81	2.81
vortex	19542	19542	266	266	324	324	2.82	2.82
twolf	8888	8888	342	343	506	506	2.12	2.16
mesa	13415	13420	1398	1276	2260	2133	1.4	1.32
equake	631	631	1	1	2	2	1.0	1.0
ammp	3995	3995	114	114	333	333	1.82	1.82
fuzz1	119	119	17	17	17	15	1.16	1.14
fuzz2	404	404	115	115	74	67	1.26	1.23
fuzz3	1515	1514	456	460	268	231	1.44	1.39

Table A.2: Register allocation statistics for Arm (Soft Float)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

	ai	vpr	gcc	crafty	gap	vortex	twolf
arm_hard	-0.02%	-0.95%	+0.04%	+0.03%	+0.00%	-0.00%	+0.00%
arm_soft	+0.03%	-0.16%	+0.04%	+0.03%	+0.00%	-0.00%	+0.00%

	mesa	equake	ammp	fuzz1	fuzz2	fuzz3	
arm_hard	-1.97%	-0.05%	+0.02%	-8.38%	-8.78%	-9.60%	
arm_soft	-0.65%	-0.04%	+0.02%	-0.36%	+0.22%	+0.39%	

Table A.3: Changes in generated code size between CompCert and CompCert^p

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.