Modeling Register Pairs in CompCert

Alexander Loitzl✉* and Florian Zuleger✉

TU Wien, Vienna, Austria
alexander.loitzl@student.tuwien.ac.at
florian.zuleger@tuwien.ac.at

Abstract. The CompCert C compiler is a moderately optimizing, formally verified compiler that ensures the preservation of the input program's semantics through a machine-checkable correctness proof. We introduce CompCert^p, an extension of the CompCert compiler, which incorporates the modeling of register pairs. This enhancement targets 32-bit architectures, such as the 32-bit Arm, which combine two registers to support 64-bit operands. So far, CompCert abstracts register pairs as 64-bit registers and allocates the entire pair when operating on 32-bit values, effectively cutting the number of available registers for 32-bit computations in half. This creates a harder register allocation problem and the emitted code requires post-processing outside of the formally verified compiler to comply with calling conventions. Our enhancement models all of Arm's registers, improving register allocation and the generated code's size. Additionaly, it models the correct calling conventions for floating-point arguments within the formal semantics, eliminating the need for unverified modifications and therefore decreasing the trusted computing base (TCB). We adapt the proofs for all CompCert-supported architectures and demonstrate that, despite a slight increase in compile time, CompCert^p generates code that is either smaller or comparable in size to that produced by the original CompCert.

Keywords: program verification · compilers · semantics · register allocation · CompCert · Coq.

1 Introduction

CompCert has gained traction in both academia [1,2,4,5,18] and industry, where it is used for the development of safety-critical systems [6,10]. The CompCert toolchain supports a large subset of C99 and several target architectures. Its correctness proof covers the translation from CompCert C, the C semantics formalized in Coq, to the target's formalized assembly language. It ensures the correctness of all optimizations and therefore rules out a common source of bugs in compilers [19].

* This work was conducted as part of the author's master's thesis project in cooperation with AbsInt.

Both the formal C semantics and that of the target machine are part of the TCB. The latter contains a model of memory, instruction semantics and the register file. Discrepancies between the model and the instruction set architecture (ISA) decrease the trust in the proof and may require unverified post-processing, as is the case for the Arm target.

The floating-point register file of 32-bit Arm (VFPv2, VFPv3-D16) is split into thirty-two 32-bit registers `S0-S31`, grouped into aligned *register pairs* `D0-D15`, as depicted in Fig. 1. Similarly, for the currently unsupported TriCore architecture, two aligned registers may be used to hold any 64-bit value.

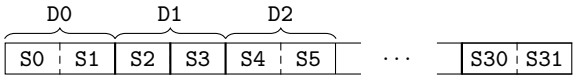

Fig. 1: Arm's register layout

Since CompCert models the register file as a simple map from registers to values, it models only the view of the combined 64-bit registers and the two halves of a pair are not adressable. This makes it possible to generate code for the Arm target, but requires additional, unverified post-processing to adhere to calling conventions. This workaround is not feasible for TriCore, as, unlike Arm, it does not have distinct floating-point registers and registers are more scarce.

Our contributions address this shortcomings by introducing register pairs in the compiler backend. While our contributions are tightly integrated into the CompCert development, we refer to the adapted version as CompCert^p for sake of clarity. Below, we highlight CompCert's shortcomings for Arm by example.

Example. Consider the C code on the left of Fig. 2. It calls the function `sum`, which sums up three floating-point numbers. The parameters a and c are single-precision (we write *single* instead of *float* to make the distinction clear) and their sum is explicitly cast to a double-precision value before adding b.

```
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;
}
```

Fig. 2: Simple program summing up floating-point numbers

The corresponding code on the right of Fig. 2 is in the RTL intermediate language of the compiler and closely resembles the original C code. RTL is the last intermediate language before register allocation, during which temporaries `x1-x6` are replaced by hardware registers. Up to this point, there is no difference between CompCert^p and CompCert.

In Fig. 3 we show excerpts of the assembly output of both CompCert^p and CompCert. The output of CompCert^p on the left directly corresponds to the instructions of the RTL code except that it uses hardware registers instead of temporaries. In the output of CompCert on the right, two additional move instructions are highlighted (line 2 and line 13), which are inserted by the aforementioned unverified post-processing step implemented in OCaml.

```
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   ...
```

```
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 s1, s4
14   bl  sum
15   mov  r0, #0
16   ...
```

Fig. 3: Assembly output of CompCert^p (left) and CompCert (right)

The aditional moves are inserted to fix the discrepancies in CompCert's internal calling conventions for the Arm hard-float target. CompCert simply passes the arguments in order, using a 64-bit register for each. In Fig. 4, we depict CompCert's allocation for the `sum` function where the two 32-bit parameters `a` and `c` block an entire 64-bit register. Arm's actual hard-float calling conventions for non-variadic functions, however, pass each argument in the first available register that fits its size. In particular, arguments are not necessarily passed in order, if, due to alignment of a double-precision argument, a single-precision register is left free.

This discrepancy is fixed by the inserted move instructions, putting each argument into the correct register: For the `sum` function, the third parameter `c` is moved from register `S4` to `S1`. We note that this strategy potentially requires more registers than Arm's calling conventions, as is the case for the `sum` function. If CompCert prematurely runs out of the eight reserved registers for passing

arguments, it may not follow the calling conventions at all and pass additional arguments on the stack instead [12]. We finally note that CompCert^p directly passes the arguments as required by Arm's calling conventions, compare Fig. 4, eliminating the need for an (unverified) post-processing step.

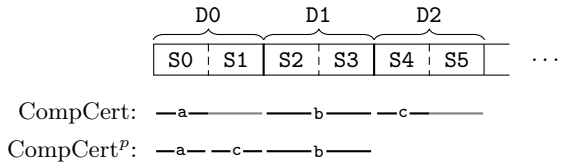

Fig. 4: Argument locations of CompCert and CompCert^p for `sum`

Contributions. We present CompCert^p, an extension of the CompCert compiler which

- improves on CompCert's Arm semantics by modeling all registers and reduces the TCB by implementing the correct calling conventions in the formal semantics,
- extends the backend to handle register pairs allowing for future support of the TriCore architecture,
- generates identical code for all other architectures and keeps their semantics unchanged,
- either improves on or performs similarly to CompCert in terms of generated code size, and
- is scheduled to be integrated into the commercially available version of CompCert distributed by AbsInt[1].

Structure. In Section 2, we give a brief overview of CompCert and its register allocator. In Section 3, we motivate our design choices and discuss their impact on the semantics and the associated proof effort in Section 4. We evaluate CompCert^p on selected benchmarks in Section 5 and conclude the paper in Section 6.

2 Overview of the CompCert Compiler

CompCert is a formally verified, optimizing C compiler supporting several target architectures (Arm, PowerPC, x86, RISC-V). It is implemented using the Coq proof assistant so that the correctness proof can be carried out directly on its source code. The proof spans the translation from CompCert's C semantics through 8 intermediate languages down to the modeled semantics of the target machine. It connects the input program's behavior to that of the assembly output via the notion of *semantic preservation.*

[1] https://www.absint.com

Definition 1 (Semantic Preservation [12]).
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, according to the definition above, CompCert may abort, e.g., on malformed input, and it may *improve* the observable behavior by giving meaning to undefined programs, which is necessary to perform dead code elimination.

2.1 Architecture of CompCert

CompCert is split into eight intermediate languages: three in the C-specific frontend [7], five in the backend [11]. To reduce the proof burden, large parts of the backend are designed to support all target architectures and the architecture specific components are plugged in. All languages share a common memory model [13] and value representation, and those languages operating on machine registers adopt a similar view of the register file.

Values. All of CompCert's languages manipulate *values*:

$$\texttt{val} \ni v \coloneqq \texttt{int}(n) \mid \texttt{long}(n) \mid \texttt{single}(n) \mid \texttt{double}(n) \mid \texttt{ptr}(b, ofs) \mid \texttt{undef}.$$

Only the arithmetic values capture their bit representation. Pointers consist of a memory block b and an offset ofs into that block. This allows pointer arithmetic within a block, while keeping pointers abstract is a requirement for reasoning about CompCert's complex memory transformations. Arithmetic values are typed naturally as `int`, `long`, `single`, and `double`, pointers are typed either as `int` or `long`, depending on the target architecture, and `undef` is of any type.

Register File and Register Allocation. CompCert adopts a simple view of the register file for its intermediate languages. It assumes a map from general purpose registers (GPRs) to values, therefore satisfying two useful properties about their read/write behavior:

$$\forall p, v: \ rs[p \leftarrow v](p) = v, \ \textit{and} \tag{1}$$
$$\forall r, r', v: \ \textit{If} \ r \neq r', \ \textit{then} \ rs[r' \leftarrow v](r) = rs(r). \tag{2}$$

The two properties are used extensively in CompCert's correctness proofs to reason about the state of the register map throughout the execution of the program.

The simple model of the register file also allows CompCert to use a textbook implementation of IRC [9]. IRC is a graph-coloring register allocator and connects the two intermediate languages RTL and LTL. The source language RTL operates on *temporaries*, of which, similar to local variables in C, an unbounded

number is available. The target language LTL operates on the limited number of available machine registers. The task of the allocator is to map the temporaries used in RTL to machine registers such that the semantics of the original program is preserved. The allocator reduces the problem of register allocation to that of graph coloring, and tries to color the *interference graph* with the available machine registers using a heuristic approach. The interference graph connects any two temporaries that have overlapping live ranges and therefore cannot be mapped to the same register. If no coloring can be found, the allocator *spills* a temporary by storing it on the stack and inserting appropriate loads and stores. Additionally, the allocator tries to delete move instructions between two temporaries by *coalescing* the source and destination temporary, making the move redundant.

3 Detailed Design Overview

The CompCert development is quite mature and proposed additions need to be carefully assessed. We focused on four key aspects: (1) the impact on the semantics of CompCert C and supported architectures other than Arm, (2) maintainability and ease of integration into the CompCert development, (3) the proof effort required to establish its correctness, and (4) its similarity to the actual machine semantics.

Fig. 5 highlights those components of CompCert that have been adapted in CompCert^p. Most importantly, we do not require any additional translation passes or intermediate languages and only three of the existing intermediate languages are affected. By keeping the interface to the machine operations (datatype `Op`), the memory model, and value representation the same, we do not change the CompCert C semantics or any of the assembly semantics other than Arm. In the following, we discuss how we achieved this by working our way backwards starting at the assembly semantics.

3.1 Assembly Semantics

Previously, in CompCert, the simple register model required all registers to be disjoint and therefore only the view of the double-precision registers was modeled. This effectively cuts the number of available registers for single-precision computations in half and the (hard-float) calling conventions could not be modeled correctly in the Arm semantics.

CompCert^p models all 32 single-precision registers rather than the 16 double-precision registers. Register pairs are explicitely modeled as pairs of their two subregisters and the register `D<n>` is refered to as `(S<2n+1>, S<2n>)`. Every access to a register pair splits and combines the value when storing and loading from the register file, respectively. Therefore, all registers are available during register allocation and we can model the correct calling conventions in the formal semantics, allowing us to omit the unverified post-processing step that inserts moves for the Arm hard-float target.

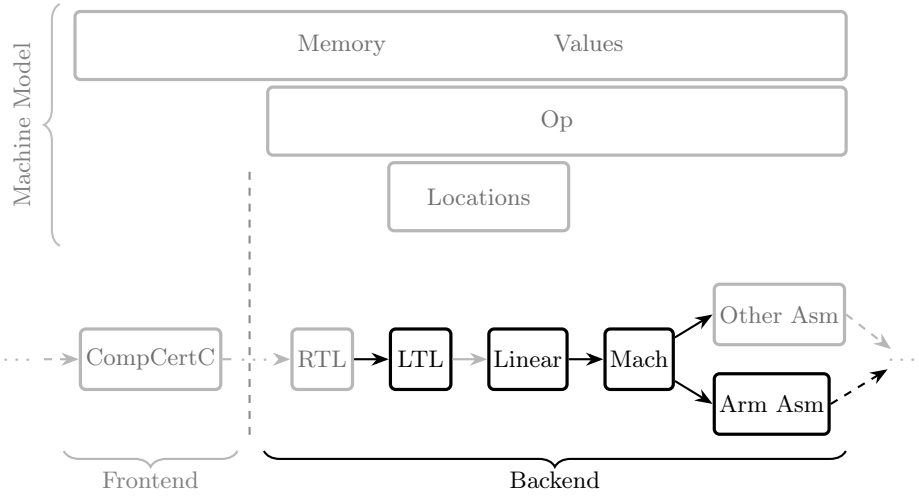

Fig. 5: Overview of CompCert with affected components highlighted

3.2 Shared Backend

To support generating code for the revised Arm target using register pairs, we need to adapt all intermediate languages after register allocation. We change the semantics of LTL, Linear, and Mach to operate on the type `rpair mreg` rather than machine registers `mreg`.

$$\texttt{rpair mreg} \ni p \coloneqq \texttt{Two}\ (r1 : \texttt{mreg}, r2 : \texttt{mreg}) \mid \texttt{One}\ (r : \texttt{mreg}).$$

The semantics are adapted such that when encountering a register pair `Two` $(r1, r2)$, the value contained in the pair is constructed from the values contained in the individual halves $r1$, and $r2$. Registers wrapped by the constructor `One` are treated as before and the register allocator, when assigning a single register to a temporary, wraps it's result with the `One` constructor. This ensures that the affected intermediate languages are downwards compatible in the sense that for those architectures using no pairs, CompCert^p generates the same code. The architecture-specific translation from the Mach intermediate language rejects any code using pairs for all architectures except Arm, allowing us to keep the semantics of all other architectures unchanged.

Since we adapt the semantics of all languages similarly, adapting the translations between them is straightforward. The translation between LTL and Linear only affects the code structure and requires no changes, indicated in Fig. 5. For the translation between Linear and Mach, the main effort lies in changing the way callee-save registers are preserved.

3.3 Register Allocation

Existing Register Allocator. CompCert's allocator, IRC, uses a heuristic approach to color the interference graph using the degree of nodes in the graph. If a node has degree less than K, the number of available machine registers, it is guaranteed to be colorable and therefore picking a color can be delayed. The node is removed from the graph and the process is repeated for the remaining graph. This simplification is interleaved with *conservative coalescing* [8], which combines two non-interfering nodes that are the source and destination of a move instruction. These two simplifications are repeated until the graph is either empty or a register has to be spilled.

Fig. 6, on the right, contains IRC's interference graph of Fig. 2's `sum` function after coalescing, hence, several nodes with no interference edge between them have already been combined. The graph contains the temporaries $x1$-$x6$ and some *pre-colored* nodes `D0`-`D2` to enforce the calling conventions. At this stage, IRC would now continue with its simplification phase, removing one low-degree node after the other until the graph is empty and then assign registers to the remaining temporaries.

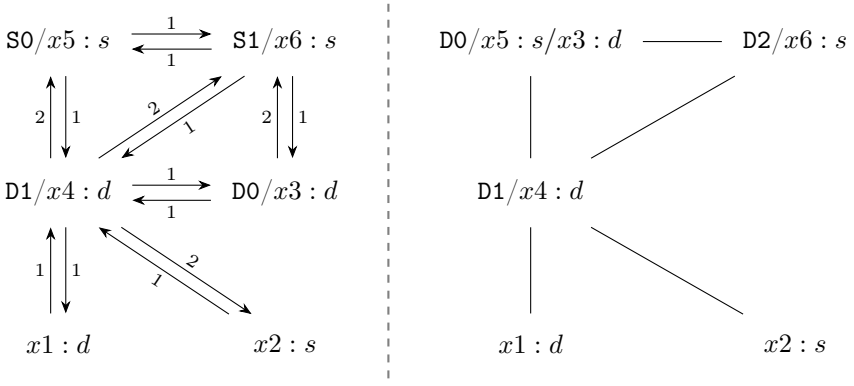

Fig. 6: Interference Graphs of `sum`. The interference graph of CompCert (right) is undirected, while that of CompCert^p (left) is a directed weighted graph. We illustrate the effect of coalescing by listing all individual nodes separated by a "/". The type of a temporary is indicated by s for single-precision and d for double-precision.

Extended IRC. The register allocator of CompCert^p is built on top of the existing implementation of IRC and closely follows the approaches presented in [16,17]. Unlike in the formal semantics, all named registers are modeled, i.e., both `D<n>` and its subregisters `S<2n>` and `S<2n+1>` are available to the allocator.

Our register allocator keeps the principal phases of IRC but captures the interactions of aliasing registers by dividing registers into classes. For Arm, we

give the allocator access to the class of integer GPRs R, the class of single-precision GPRs S, and the class of double-precision GPRs D. We define the measure $worst(C, C')$ between any two classes C and C'. The measure 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 register candidates. If registers of two classes cannot interfere, e.g., an integer register with a floating-point register, we leave $worst$ undefined. All architectures except Arm have two classes, floating-point and integer GPRs, and we define $worst(C, C') = 1$ if $C = C'$, and otherwise leave it undefined.

We can capture the new interactions by using a weighted directed graph. 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. Hence, the weight of an edge $e = (u, v)$ captures the number of v' register candidates blocked by the source u. The degree $d(v)$ of a node v is the sum of the incoming edge weights:

$$d(v) = \sum_{e=(u,v)\in E} \omega(e). \tag{3}$$

Since IRC caches the degree of all nodes and updates them whenever the graph is transformed, we keep the internal representation of the non-directed graph, but use $worst$ to update the cached degrees.

Consider CompCert^p's interference graph on the left of Fig. 6. The edge from $\texttt{D1}/x4$ to $x2$ has weight 2 as it blocks two candidates (`S2` and `S3`) for a potential coloring of $x2$. The principal idea of coalescing and the simplification phase is then the same as before where nodes of low degree can be safely removed. During the actual coloring, aliasing has to be respected, ensuring that two neighboring nodes are not colored with two overlapping registers.

4 Semantics and Proof Effort

Our extensions are built on top of CompCert and therefore all proofs have been carried out using the Coq proof assistant. We make the development available as an artifact [14] and only comment on the most interesting problems. As illustrated in Fig. 5, the changes are limited to a handful of intermediate languages and the Arm assembly semantics. Adapting the semantics of all languages similarly, splitting and recombining at the level of values, allows for a straightforward adaptation of CompCert's proofs of semantics preservation. Rather than CompCert^p being a standalone addition, the changes are tightly integrated into the entire compiler. In total, the development grew by around 3k lines of code and amounts to about 3 person-months of effort. Below we discuss the consequences of value splitting on the semantics, the verification of the register allocation, and the preservation of callee-save registers.

4.1 Pairs in the Verified Backend

To facilitate the correctness proofs of the translation passes, the components shared by multiple intermediate languages, like machine operations and memory, operate on values. The specifics of a language may then be captured as part of its state. Below we show a simplified version of the RTL transition rule for an arithmetic operation, only listing those parts of the state that change as part of the transition. For a more detailed description, we refer to [11].

$$\frac{c(pc) = \lfloor \texttt{op}_{RTL}(op, \vec{r}, r, pc') \rfloor \qquad \texttt{eval\_op}(\_, \_, op, R(\vec{r})) = \lfloor v \rfloor}{\_ \vdash S(\_, \_, \_, pc, R, \_) \xrightarrow{\epsilon} S(\_, \_, \_, pc', R[r \leftarrow v], \_)}$$

An RTL execution state contains the current program counter pc and a map R from temporaries to values. The rule defines the conditions in order to take a step in the RTL semantics, if the current program counter pc is pointing to an $\texttt{op}_{RTL}$ instruction. If performing operation op on the values contained in the registers of the arguments $\vec{r}$ evaluates to v, we can take a step, updating the result temporary r with value v.

We have a similar rule for LTL where the instruction $\texttt{op}_{LTL}$ operates on optional register pairs (`rpair mreg`) and the state contains a map L from locations (Stack slots and registers) to values. We extend the location map in a straightforward fashion by defining three operations: `combine`, `loword`, and `hiword` to perform splitting and combining on the level of values.

$$L((r1, r2)) \stackrel{def}{=} \texttt{combine}(L(r1), L(r2)), \ \textit{and}$$
$$L[(r1, r2) \leftarrow v] \stackrel{def}{=} L[r1 \leftarrow \texttt{hiword}(v)][r2 \leftarrow \texttt{loword}(v)].$$

Below, we show the transition rule for $\texttt{op}_{LTL}$, which is similar to that of RTL, but captures the execution in a basic block, and therefore does not increase the program counter. Note however, that the arguments $\vec{p}$ and result location p are either single registers or pairs and are evaluated as given above.

$$\frac{\texttt{eval\_op}(\_, \_, \texttt{op}, L(\vec{p})) = \lfloor v \rfloor}{\_ \vdash B(\_, \_, \_, \texttt{op}_{LTL}(op, \vec{p}, p) :: bb, L, \_) \xrightarrow{\epsilon} B(\_, \_, \_, bb, L[p \leftarrow v], \_)}$$

Proving that an execution of the $\texttt{op}_{LTL}$ instruction preserves the semantics of an associated $\texttt{op}_{RTL}$ instruction breaks down to showing that we pass the same values to the operation op, i.e., $R(\vec{r}) = L(\vec{p})$. Hence, we need to show that we can correctly use the operations `combine`, `hiword`, and `loword` to split and reconstruct values. While splitting and combining the bit-representation of data is intuitive, we cannot directly model it with CompCert's value representation.

Splitting Values. We cannot give meaning to the splitting and combining of arbitrary values. We define `hiword` and `loword` only on `long` and `double` values and the `combine` operation is only defined for two `int` or two `single` values. We use the constructor `single` to be able to differentiate between split halves of `double` and `long` values. The three operations allow us to split and recombine values of type `double`, captured by Lemma 1.

Lemma 1. $\forall v, \vdash v : \texttt{double} \Rightarrow v = \texttt{combine}(\texttt{hiword}(v), \texttt{loword}(v))$.

We can state a similar lemma for `long` with the additional requirement, that for the current target machine, pointers do not have type `long`. Since pointers do not expose their bit representation, we cannot split them.

Lemma 1 lets us reason about recombining a value that has been previously split, hence allowing us to reason about loading from the register file after a store. The converse is not possible, i.e., we cannot guarantee anything about a value that has been combined with another and then split again. Even if type information is available, since `undef` is of any type, combining arbitrary values might destroy any information.

4.2 Validating the Allocation Result

The register allocation is performed by untrusted OCaml code. Only the resulting LTL code is checked for semantic preservation of the original RTL code by the formally verified validation algorithm described in [15].

CompCert^p's LTL semantics is the first that allows pairs as operands and the new validator needs to be able to relate the original RTL code, which stores double-precision values in temporaries, to the LTL code, which splits and combines the values stored in pairs. The updated validation algorithm retains the two phases of the original algorithm, first performing structural checks relating the two programs syntactically and then a data-flow analysis relating live temporaries with their allocated register or stack slot.

Our validation algorithm restricts accepted LTL programs to those in which pairs are only used to hold values that can be correctly split and recombined. During the structural checks, we additionally check that the subregisters of a pair are distinct and ensure that, if the operands of a move are pairs, they are disjoint. The data-flow analysis relates temporaries to the machine registers they have been mapped to. If mapped to a pair, we ensure that it contains a value of type `long` or `double`. This allows us to make use of Lemma 1, or the corresponding lemma for `long`, to reason about the result of loading the pair from the location map, relating it to the original value in the RTL code. Going back to the `op` instructions above, we can then prove that, if the two sets of arguments, $\vec{r}$ and $\vec{p}$, are related by the data-flow analysis, we load the same values from the respective maps.

4.3 Preservation of Callee-Save Registers

Up to the Mach intermediate language the automatic preservation of callee-save registers is built into the semantics of function calls. 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 ensure their preservation across function calls. The new instructions do not have any corresponding ones in the previous translation passes,

setting the Stacking pass apart from other translation passes affected by our changes.

When reasoning about the preservation of callee-save registers, there is no information about their contained values at compile time. This prevents us from storing and loading pairs as we cannot establish correct splitting and recombining of values. The naive solution of just storing each single-precision register individually might double the required load and store instructions.

```
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.
```

Fig. 7: Pseudocode of `save_callee_rpair` for pairs

To resolve this issue, we introduce two new instructions to the Mach intermediate language to perform stack accesses. Unlike the other instructions, `Msavecallee` and `Mrestorecallee` do not combine and split values but rather treat the two halves individually. Fig. 7 showcases the semantics of `Msavecallee` for a register pair. We write $m[(sp, ofs) \leftarrow_\kappa v]$ to denote storing a value v of type κ at the location (sp, ofs) in memory m.

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 each half at its correct address. In addition, we need to explicitly compute the offset to store the second register.

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 in Fig. 8. It is quite similar to `save_callee_rpair` and has a matching `contains` predicate for each store. 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.

Equipped with the `contains_rpair` predicate, Fig. 9 states the lemma capturing the use of `save_callee_rpair` to save a register pair to the stack. As expected, the lemma allows us to establish that, after saving a register to the stack using `callee_save_rpair`, the memory satisfies the `contains_rpair` predicate, which states that the memory contains the two values stored in the register pair.

```
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.
```

Fig. 8: Simplified separating conjunction `contains_rpair`

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 m sp ofs P 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.
```

Fig. 9: Lemma establishing the correct saving of callee-save registers

When proving the correct preservation of the callee-save registers we rely on the properties and functions showcased above. We can state and prove a similar lemma that allows us to correctly reload the values from the stack given it satisfies the separating conjunction.

The changes to the Mach semantics are also 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`, respectivley. The instructions have already been introduced to CompCert to handle the loading and storing of callee-save registers. They reflect the semantics of `Msavecallee` and `Mrestorecallee`, now similarly storing the individual registers and explicitly computing the offset. This is defined in accordance with the Arm semantics [3].

5 Experimental Evaluation

The development of CompCert^p builds on CompCert (Release 23.10) distributed by AbsInt and we compare CompCert^p against this release. For a fair comparison, we additionally include a small fix in the unverified post-processing[2], which we discovered during the development of CompCert^p. The artifact [14] published online contains all proofs on the public version of CompCert.

We compare generated code size, compile time, and allocation statistics of CompCert and CompCert^p. We take several benchmarks from the SPEC CPU 2000 [3] benchmark suite and three sets of 100 generated C code files: *fuzz1*, *fuzz2*, and *fuzz3*. They contain increasingly complex floating-point expressions and function calls using floating-point parameters. We use them as a stress test of our register allocator to highlight the changes in the code generation of CompCert^p and include them in the artifact.

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 uses `D0 - D7` (`S0 - S15`) to pass floating point parameters as described previously. The soft float ABI uses the standard calling conventions to pass arguments in the registers `R0-R3` and only uses the floating-point registers for computations. All tests were run 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*. Below we highlight the most interesting changes and include the complete data in the appendix.

5.1 Compile Time

Fig. 10 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 show the standard deviation with 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, we see an increased time for both register allocation and allocation validation.

In Table 1, we capture the average change in compile time across all benchmark suites for all tested targets. We see a slight increase in total compile time and for Arm a significant increase in the time it takes to validate the allocation result. Since allocation validation is usually short, this does not contribute a big increase to the total compile time.

[2] https://github.com/AbsInt/CompCert/commit/ccb88a8
[3] https://www.spec.org/cpu2000
[4] https://perf.wiki.kernel.org

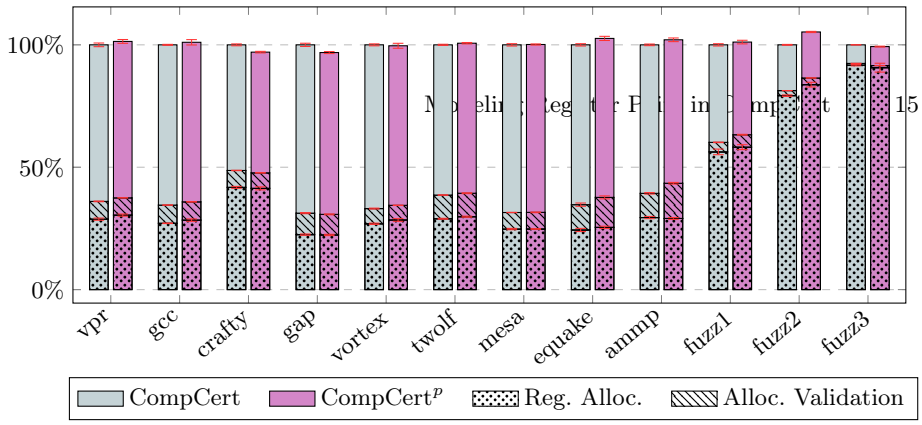

Fig. 10: Relative compile times for Arm (hard float).

Table 1: Relative change in compile times split into phases

Phase	arm_hard	arm_soft	riscv
Register Allocation	+2.41%	+2.0%	+4.00%
Allocation Validation	+10.94%	+11.66%	+1.17%
Remaining Translations	-1.43%	-0.55%	+0.59%
Total compile time	+0.59%	+0.54%	+1.91%

5.2 Code Size

Since many of the SPEC benchmarks do not operate on floating-point values, code size does not change significantly for most of the contained test suites. Hence, we only report on the *vpr* and *mesa* benchmarks, which contain a significant number of floating point operations, and our generated *fuzz* benchmarks.

Table 2 contains the change in the generated code size of CompCert^p in relation to CompCert. 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 significantly decreases code size. For the soft float target, we see a smaller impact and even a slight increase in code size for some benchmarks. The soft float ABI only uses four 32-bit registers to pass arguments and we suspect this bottleneck to decrease the positive effects of the additional available registers. Since it does not use the previously unmodeled single-precision registers to pass arguments, it does not benefit from the omission of move instructions.

Table 2: Relative change in generated code size for Arm

ABI	vpr	mesa	fuzz1	fuzz2	fuzz3
hard float	-0.83%	-1.77%	-4.78%	-4.7%	-4.7%
soft float	-0.2%	-0.71%	-0.2%	+0.19%	+0.27%

5.3 Allocation Statistics

We list detailed allocator statistics for those benchmarks for which we recorded the code size in Section 5.2. We capture the moves remaining in the program after register allocation and the moves inserted by the unverified pass for the hard float ABI. Additionally, we record the number of reloads and spills inserted by the allocator. For the fuzzed suites we take the average of all contained test cases.

Table 3 contains the statistics for the hard float ABI. We can see that the post-processing phase of CompCert, necessary to comply with Arm's calling conventions, inserts a considerable amount of moves, whereas this is not necessary in CompCert^p. In all benchmarks, except for *vpr*, we recorded a considerable reduction in spills or reloads. Note, that we do not count reloads that happen as part of enforcing the calling conventions.

Table 3: Selected register allocation statistics for Arm (hard float)

Instance	Remaining		Inserted		Spills		Reloads	
	C	C^p	C	C^p	C	C^p	C	C^p
vpr	4557	4557	165	0	275	275	298	297
mesa	13414	13420	939	0	1401	1276	2265	2133
fuzz1	119	118	40	0	17	17	17	15
fuzz2	404	404	148	0	115	115	74	65
fuzz3	1515	1515	533	0	456	461	267	226

For the soft float ABI, the difference between CompCert and CompCert^p is less significant. For the *vpr* benchmark, their output is similar and for the other affected benchmarks, the improvements are smaller. For details we refer to the appendix.

5.4 Summary

To sum up, CompCert^p generally improves on CompCert in terms of code generation. It either generates similar or smaller code if floating-point calculations are performed by the input program. The improvement comes from the omission of move instructions inserted by CompCert's unverified post-processing and a decrease in spill code. This comes at the cost of a slightly increased compile time for all architectures.

6 Conclusion

CompCert^p's adapted backend supports register pairs such that the semantics of architectures with no pair 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. We can therefore omit a previously required, unverified post-processing step of CompCert, thereby increasing the trust in the correctness proof. With support for register pairs in the backend of the compiler, we are now able to support a TriCore backend which is under development.

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.

Acknowledgments. We thank Bernhard Schommer, Michael Schmidt, and Christoph Mallon at AbsInt for valuable discussions about CompCert and register allocation.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Appendix

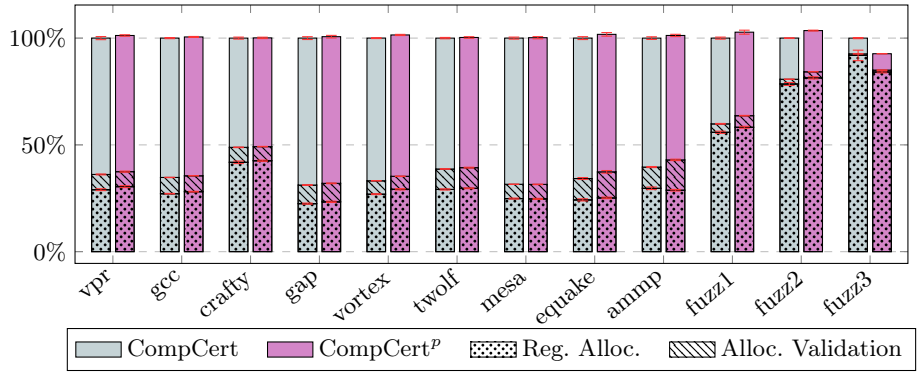

Fig. 11: Relative compile time: Arm (soft float).

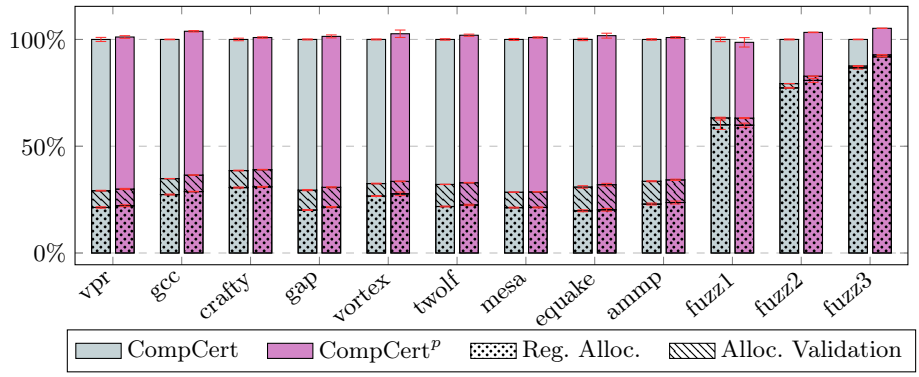

Fig. 12: Relative compile time: RISC-V.

Table 4: Register allocation statistics for Arm (hard float)

Instance	Remaining		Inserted		Spills		Reloads	
	C	C^p	C	C^p	C	C^p	C	C^p
vpr	4557	4557	165	0	275	275	298	297
gcc	52810	52810	0	0	1125	1125	2006	2006
crafty	19501	19501	0	0	512	512	240	241
gap	20145	20145	0	0	390	390	869	869
vortex	19542	19542	0	0	266	266	324	324
twolf	8888	8888	0	0	342	342	506	506
mesa	13414	13420	939	0	1401	1276	2265	2133
equake	631	631	0	0	1	1	2	2
ammp	3995	3995	0	0	114	114	333	333
fuzz1	119	118	40	0	17	17	17	15
fuzz2	404	404	148	0	115	115	74	65
fuzz3	1515	1515	533	0	456	461	267	226

Table 5: Register allocation statistics for Arm (soft float)

Instance	Remaining		Spills		Reloads	
	C	C^p	C	C^p	C	C^p
vpr	4557	4557	275	275	298	297
gcc	52810	52810	1125	1125	2006	2006
crafty	19501	19501	512	512	240	241
gap	20145	20145	390	390	869	869
vortex	19542	19542	266	266	324	324
twolf	8888	8888	342	342	506	506
mesa	13415	13420	1398	1276	2260	2133
equake	631	631	1	1	2	2
ammp	3995	3995	114	114	333	333
fuzz1	119	119	17	17	17	16
fuzz2	403	404	115	116	75	68
fuzz3	1515	1562	456	472	268	242

References

1. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: Certicoq: A verified compiler for coq. In: The third international workshop on Coq for programming languages (CoqPL) (2017)
2. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) Programming Languages and Systems. pp. 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1

3. Arm Limited: Arm architecture reference manual for a-profile architecture, https://developer.arm.com/documentation/ddi0487
4. Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., Trieu, A.: Formal verification of a constant-time preserving c compiler. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371075
5. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an ssa-based middle-end for compcert. ACM Trans. Program. Lang. Syst. **36**(1) (mar 2014). https://doi.org/10.1145/2579080
6. Bedin França, R., Blazy, S., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Formally verified optimizing compilation in ACG-based flight control software. In: ERTS2 2012: Embedded Real Time Software and Systems. AAAF, SEE, Toulouse, France (Feb 2012)
7. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 460–475. Springer (2006). https://doi.org/10.1007/11813040_31
8. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. **16**(3), 428–455 (may 1994). https://doi.org/10.1145/177492.177575
9. George, L., Appel, A.W.: Iterated register coalescing. ACM Trans. Program. Lang. Syst. **18**(3), 300–324 (1996). https://doi.org/10.1145/229542.229546
10. Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In: ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems. pp. 1–9. 3AF, SEE, SIE, Toulouse, France (Jan 2018)
11. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
12. Leroy, X.: The compcert c verified compiler: Documentation and user's manual (2023)
13. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model. In: Appel, A.W. (ed.) Program Logics for Certified Compilers. Cambridge University Press (Apr 2014). https://doi.org/10.1017/CBO9781107256552.037
14. Loitzl, A.: Artifact for "Modeling Register Pairs in CompCert". https://doi.org/10.5281/zenodo.12010656
15. Rideau, S., Leroy, X.: Validating register allocation and spilling. In: Gupta, R. (ed.) Compiler Construction. pp. 224–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11970-5_13
16. Runeson, J., Nyström, S.O.: Retargetable graph-coloring register allocation for irregular architectures. In: Krall, A. (ed.) Software and Compilers for Embedded Systems. pp. 240–254. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39920-9_17
17. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. SIGPLAN Not. **39**(6), 277–288 (2004). https://doi.org/10.1145/996893.996875
18. Song, Y., Cho, M., Kim, D., Kim, Y., Kang, J., Hur, C.K.: Compcertm: Compcert with c-assembly linking and lightweight modular verification. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371091
19. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 283–294. PLDI '11, Association

for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993532