An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code

Yuting Wang[1], **Pierre Wilke**[1,2], Zhong Shao[1]

Yale University[1], CentraleSupélec[2]

POPL'19 – January 18th, 2019

Verified compilation

CompCert : verified C compiler (Leroy *et al.*, first released in 2008)

Used as a basis for a large number of extensions:

- alternate semantics: CompCertTSO (weak memory model, Sevcík *et al.*, JACM'13), CompCertS (undefined pointer arithmetic, Besson *et al.*, ITP'17)
- a more concrete view of the stack: Quantitative CompCert (merge the stack blocks into a single stack region, Carbonneaux *et al.*, PLDI'14)
- compositional compilation: Compositional CompCert (Stewart *et al.*, POPL'15), compositional semantics (Ramananandro *et al.*, CPP'15), SepCompCert (Kang *et al.*, POPL'16)

Open problems:

- verified compilation to machine code
- port all compiler passes of CompCert, including challenging inlining and tailcall recognition
- verified compilation of heterogeneous modules (mix C and Asm modules)

Contribution: Stack-Aware CompCert

A version of CompCert with:

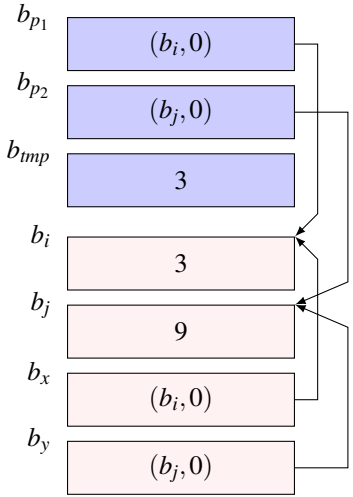
1. compilation to machine code
   - merge the stack blocks into a unique stack region
   - eliminate CompCert's pseudo-instructions
   - generate machine code

2. complete extension: we support all CompCert passes
   - including challenging optimizations (function inlining, tailcall elimination)

3. compositional compilation
   - stack access policy
   - mix C and Asm programs

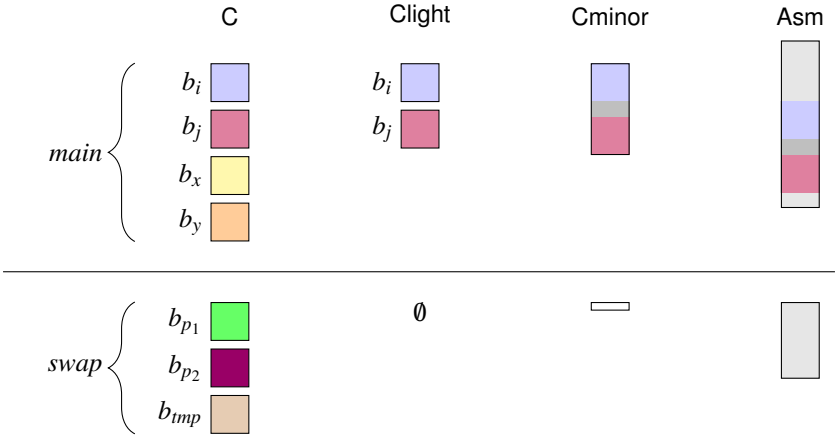
CompCert: memory model and values

```
void swap(int * p1, int * p2){
  int tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
}
int main(){
  int i = 3, j = 9;
  int * x = &i;
  int * y = &j;
  swap(x,y);
  return 0;
}
```



CompCert: compilation and memory model

The memory model stays the same throughout compilation, but the memory blocks change shapes.



The stack frames in Asm are in distinct blocks!

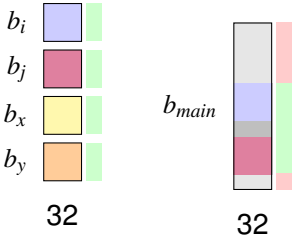
The abstract stack

We maintain an abstract stack in memory states, that reflects the structure of the concrete stack.

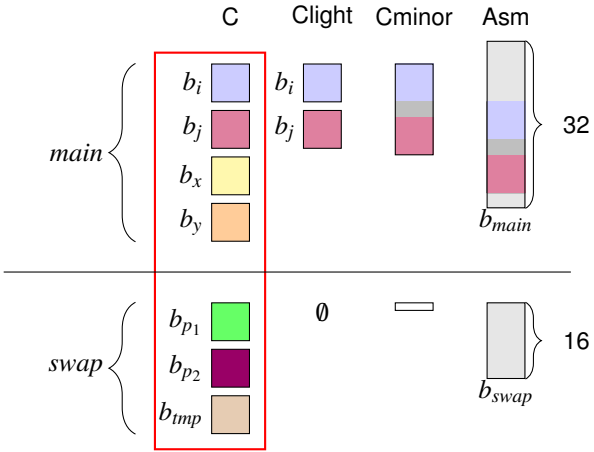
Abstract stack: a list of abstract frames.

An abstract frame records useful information about a concrete stack frame:

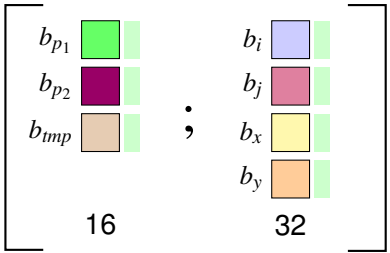
- the size of this stack frame at the assembly level;
- which blocks are part of that stack frame;
- which locations of these blocks are public or private



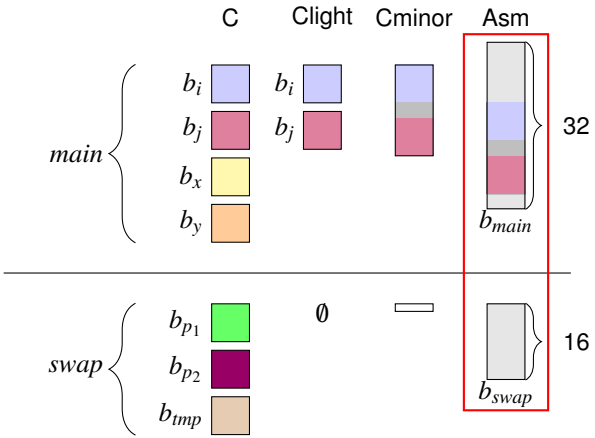
Abstract stack: example



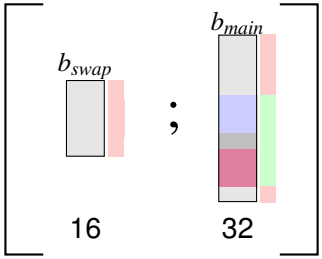
The abstract stack at the C level is:



Abstract stack: example



The abstract stack at the Asm level is:

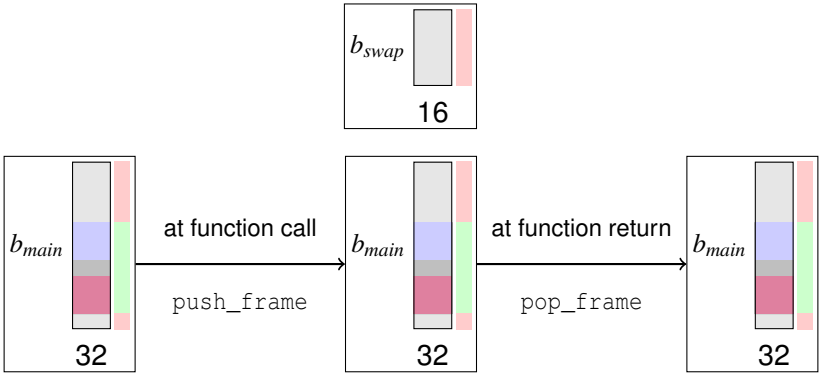


Stack-access policy: we may write to

- all of b_{swap}
- public locations in b_{main}

Abstract stack primitives

Semantics of all intermediate languages instrumented with `push_frame` and `pop_frame`

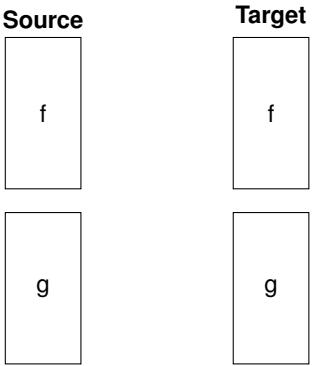


Key argument for merging stack blocks :

The `push_frame` primitive only succeeds if the sum of the frames' sizes is lower than `MAX_STACK`.

Preservation of stack usage with compilation

Since the semantics include stack consumption, it must be preserved by compilation
Property to ensure: at each program point, the size of source stack should be larger than (or equal to) the size of target stack.



Regular case

The sizes of the source and target stacks are equal.

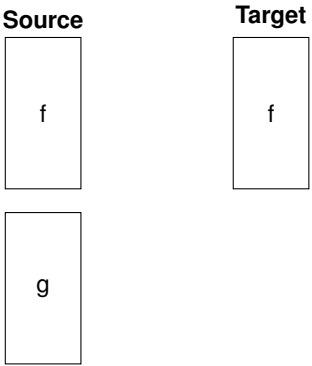
$$|f| + |g| = |f| + |g|$$

Recall $|f|$ is the size of f's stack frame at the Asm level!

Preservation of stack usage with compilation

Since the semantics include stack consumption, it must be preserved by compilation

Property to ensure: at each program point, the size of source stack should be larger than (or equal to) the size of target stack.



Function inlining

Source

```
  void g(){
⇒   G;
  }
  void f(){
    g();
  }
```

Target

```
  void g(){
    G;
  }
  void f(){
⇒   G;
  }
```

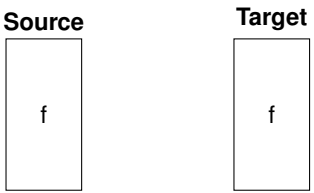
The sizes of the source stack is larger than the target stack.

$$|f| + |g| \geq |f|$$

Preservation of stack usage with compilation

Since the semantics include stack consumption, it must be preserved by compilation

Property to ensure: at each program point, the size of source stack should be larger than (or equal to) the size of target stack.



Tailcall inlining

Source

```
  void g(){
    G;
  }
  void f(){
    F;
⇒   tail g();
  }
```

Target

```
  void g(){
    G;
  }
  void f(){
⇒   F;
    G;
  }
```

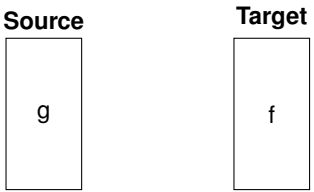
The sizes of the source stack is larger than the target stack.

$$|f| \geq |f|$$

Preservation of stack usage with compilation

Since the semantics include stack consumption, it must be preserved by compilation

Property to ensure: at each program point, the size of source stack should be larger than (or equal to) the size of target stack.



Tailcall inlining

Source

```
  void g(){
⇒   G;
  }
  void f(){
    F;
    tail g();
  }
```

Target

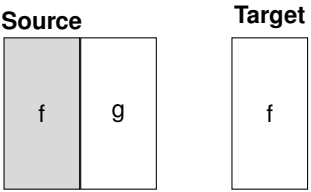
```
  void g(){
    G;
  }
  void f(){
    F;
⇒   G;
  }
```

Problem: How to compare the sizes of the source and target stacks

$$|g| \stackrel{?}{\geq} |f|$$

Preservation of stack usage with compilation

Since the semantics include stack consumption, it must be preserved by compilation
Property to ensure: at each program point, the size of source stack should be larger than (or equal to) the size of target stack.



Tailcall inlining

Source

```
  void g(){
⇒   G;
  }
  void f(){
    F;
    tail g();
  }
```

Target

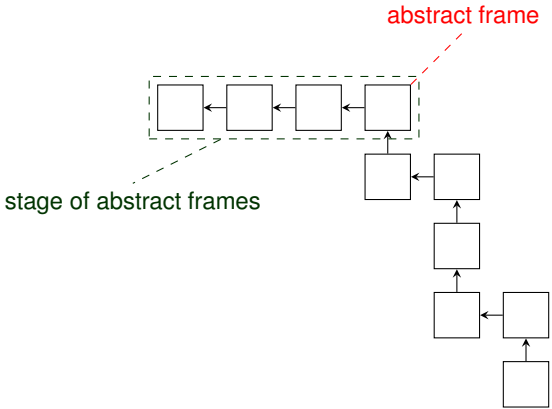
```
  void g(){
    G;
  }
  void f(){
    F;
⇒   G;
  }
```

We keep the history of tailcalled functions:

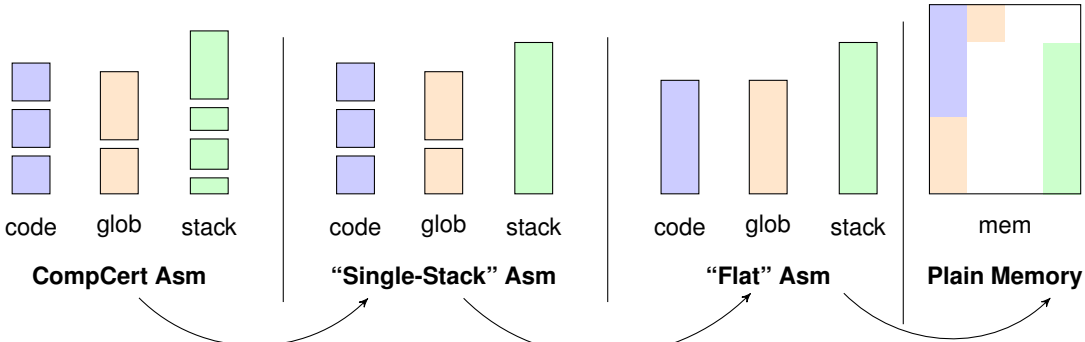
$$max(|f|, |g|) \geq |f|$$

The structure of the abstract stack

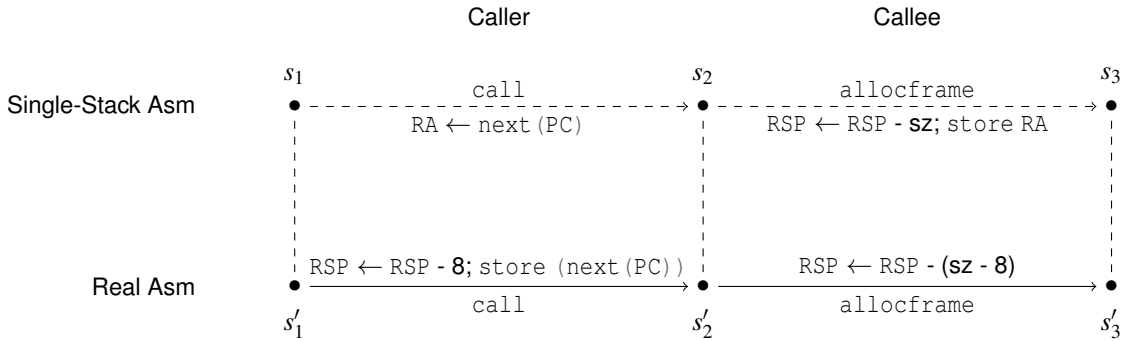
The abstract stack is actually a list of list of abstract frames.



From CompCert Assembly to Machine Code



Eliminating pseudo-instructions

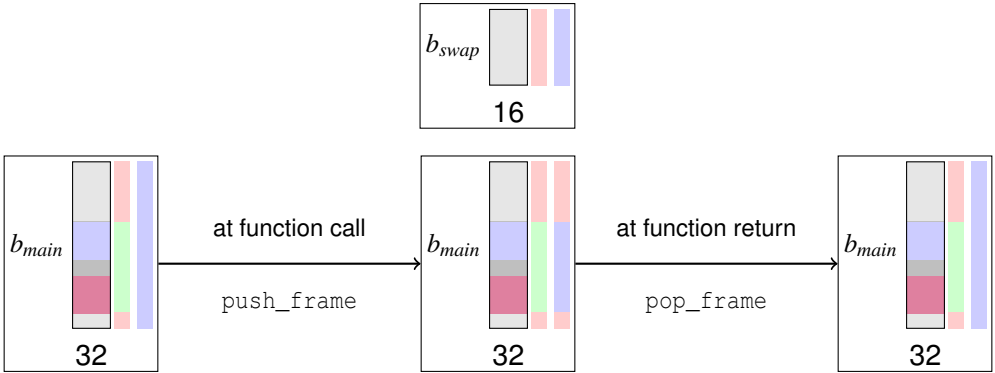


Mismatch between CompCert semantics and expected semantics

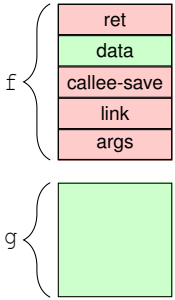
We get rid of the pseudo-register `RA` and can do away with pseudo-instructions (simple pointer arithmetic)

Stack access policy

Accessible locations are either top-frame locations or public locations.



Contextual compilation



When a function `f` calls a function `g`, the private regions of `f`'s stack frame should not be altered.

Programs compiled from C comply with that policy.

Characterization of acceptable Asm functions.

We apply this principle to CompCertX (Gu *et al.*, POPL'15)

- contextual compiler developed for CertiKOS
- ability to mix C and Asm functions

Comparison with existing work

	Target	Completeness	Compositionality	Time	LOC
CompCert(3.0.1)	CompCert Asm	complete	separate	-	135k
Stack-Aware CompCert	**Machine Code**	**complete**	**contextual**	**10.5**	**+48k**
Quantitative CompCert	SingleStack Asm	w.o. some opts.	N/A	-	100k
Compositional CompCert	CompCert Asm	w.o. some opts.	general	10	200k
SepCompCert	CompCert Asm	complete	separate	2	+3k
CompCertX	CompCert Asm	no s.a. data	contextual	-	+8k
CompCert-TSO	x86-TSO	w.o. some opts.	concurrency	45	85k
CompCertS	CompCert Asm	w.o. some opts.	N/A	25	220k

Conclusion

We develop Stack-Aware CompCert, with three distinguishing features:

1. compilation to machine code
   - finite-size stack
   - more concrete memory layout for Asm
   - closer to actual machine code: reduction of unverified part of the compiler
2. complete extension of CompCert
   - function inlining and tailcall elimination
3. compositional compilation
   - extension of CompCertX
   - stack access policy

Further work and perspectives:

- port to other backends: ARM, RISC-V, x86-64
  - main challenge: encoding and decoding of instructions
- define a stack analysis / verification framework to reason about the stack usage of programs and prove they run in bounded stack