

Scheduling for the CompCert verified compiler

Sylvain Boulmé Léo Gourdin David Monniaux Cyril Six

VERIMAG

March 30, 2021



CompCert

Formally verified C compiler

project led by Xavier Leroy, then at INRIA, now at Collège de France

Non-commercial <https://github.com/AbsInt/CompCert>

Commercial <https://www.absint.com/compcert/index.htm>

trace of execution = sequence of external calls, volatile read/writes

valid trace of execution at C level



same trace of execution at assembly level



Use case: traceability

Safe-critical systems (e.g. avionics)

Obligation to match object code to source

Conventional method: -OO and some manual inspection

CompCert replaces this by mathematical proofs.

Can **use optimization**.



KALRAY

Semantics and proofs in CompCert

Each intermediate language comes with a **semantics** written in Coq.

Optimization / transformation phases written in Coq.
(Can call external untrusted OCaml code.)

Must prove **simulation** for each phase



Simulation proofs

Lockstep

one step of program before the transformation



one matching step of program after the transformation

More complex simulations replace sequences of steps by sequences of steps.



A menu

1. oysters
2. veal blanquette
 - 2.1 prepare blanquette
 - 2.2 cook it
3. millefeuille
 - 3.1 **puff pastry**
 - 3.1.1 fold 1, wait 30 minutes
 - 3.1.2 fold 2, wait 30 minutes
 - 3.1.3 fold 3, wait 30 minutes
 - 3.1.4 fold 4, wait 30 minutes
 - 3.1.5 fold 5
 - 3.2 cream



KALRAY

Scheduling

“Official” CompCert produces instructions roughly in the source ordering.

Not the best execution order in general!

Especially on in-order cores.

Our solution: **verified scheduling**



Superblock scheduling

1. Partition each function into **superblocks**: one entry point, possibly several exit points, no cycle
2. Possibly do some other reorganization: tail duplication, etc. to get bigger superblocks
3. Schedule the superblock (no proof needed)
4. Witness through symbolic execution that the original and scheduled superblocks have equivalent **semantics** (proof needed)

Before register allocation, on IR.

On Kalray K VX and AArch64: reschedule basic blocks on assembly instructions after register allocation.



KALRAY

Equivalent semantics

- ▶ Same order of exit branches in original and scheduled superblock
- ▶ All live pseudo registers and memory have the same value at same exit point (non-live registers can differ)
- ▶ Same (or smaller) list of instructions that may fail (division by zero, memory access) reached at same exit point

Obtained by **symbolic execution**: two registers are considered equal if computed by exactly the same symbolic terms



Example

$r_1 := a * b$

$r_3 := a - b$

$r_2 := r_1 + c$

branch($a > 0$, EXIT1)

$r_3 := a - b$

$r_4 := a * b$

$r_2 := r_4 + c$

branch($a > 0$, EXIT1)

r_1 and r_4 are both dead at EXIT1 and at final point.

These two blocks are **equivalent**: in both cases

$r_2 = (a * b) + c$ and $r_3 = a - b$

Acceptable refinement

$r_1 := a * b$

$r_3 := a - b$

$r_2 := r_1 + c$

$r_5 := a/b$

branch($a > 0$, EXIT1)

r_5 dead on EXIT1.

$r_3 := a - b$

$r_4 := a * b$

$r_3 := r_4 + c$

branch($a > 0$, EXIT1)

$r_5 := a/b$

On x86, the division may fail:

- ▶ it's allowed to move it beyond the branch
- ▶ the converse is not allowed



KALRAY

Information needed

For all instructions

- ▶ **latency**: clock cycles between consuming operands and producing the value (or, more generally, a timetable of when each operand is consumed after the instruction is issued)
- ▶ **resource consumption**: CPU units in use that preclude other instructions being scheduled at the same time

Very difficult to find even for “open cores”!!!
(Reverse-engineer gcc and LLVM?)



KALRAY

Current work in progress

Rewrite instructions sequences into equivalent sequences, on intermediate code

(Already implemented on assembly code after register allocation: rewrite multiple consecutive memory accesses into multiple load/stores, on K VX and AArch64)



Performance gain

CPU	Differences in cycles spent (%) compared to no CSE3, no unroll					
	avg	min	max	gcc -O2	min	max
Cortex-A53	-16	-63	+3	+10	-23	+87
Rocket	-10	-43	+1	+29	0	+184
Xeon	-21	-56	+4	+21	-3	+189
KV3	-11	-32	+3	+8	-13	+88



KALRAY

Suggested instruction: conditional move

Wanted by companies that want predictable hard real time code
(fewer execution paths)

Branches are **bad** for worst-case execution time static analysis
(Absint aIT, etc.)

Suggestion: add **conditional moves for integer and floating-point registers**
at least on in-order cores



Suggestion: dismissible loads

An operation that may fail cannot be moved before a branch

$r_1 := a + i \ll 3$

branch($i > 3$, EXIT1)

$r_2 := \text{load}(p)$

$r_1 := a + i \ll 3$

$r_2 := \text{load}_s(p)$

branch($i > 3$, EXIT1)

Cannot be done if the load can fail.

Need special load returning a **default value** instead of trapping.

- ▶ easy without virtual memory
- ▶ needs OS collaboration with virtual memory



KALRAY

Dismissible load on K VX

8 cycles

L100:

```
compw.ge $r32 = $r4, $r2
;;
cb.wnez$r32? .L101
;;
sxwd $r5 = $r4
addw $r4 = $r4, 1
;;
lws.xs $r3 = $r5[$r1]
;;
addw $r0 = $r0, $r3
goto .L100
;;
```

6 cycles

.L100:

```
sxwd $r5 = $r4
compw.ge $r32 = $r4, $r2
;;
lws.s.xs $r3 = $r5[$r1]
;;
cb.wnez $r32? .L101
;;
addw $r0 = $r0, $r3
addw $r4 = $r4, 1
goto .L100
```



KALRAY

A general call for collaboration

Need collaboration between

- ▶ compiler writers
- ▶ architecture / core designers
- ▶ operating systems (low level)

Our version of CompCert with optimizations not found in the “official” releases + the K VX target:

<https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx>

Pre-pass scheduling: K VX; Cortex-A53/A35 (AArch64); Rocket, SweRV EH1 (Risc-V)

Post-pass scheduling: K VX; Cortex-A53/A35 (AArch64)

