Instruction Scheduling and Register Allocation on ARM Cortex-M

Ko Stoffelen

iCIS | Digital Security
Radboud University

Problem

How to write high-speed (assembly) code for microprocessors, when insufficient registers and slow memory loads are the bottleneck?

Problem

How to write high-speed (assembly) code for microprocessors, when insufficient registers and slow memory loads are the bottleneck?

Answer:
Proper instruction scheduling and register allocation, including efficient spill code generation

Problem

How to write high-speed (assembly) code for microprocessors, when insufficient registers and slow memory loads are the bottleneck?

Answer:
Proper instruction scheduling and register allocation, including efficient spill code generation

But how?

Old problems in CS

Instruction scheduling

Given program as CPU instructions, reorder them to minimize pipeline stalls (without changing semantics)

Old problems in CS

Instruction scheduling

Given program as CPU instructions, reorder them to minimize pipeline stalls (without changing semantics)

Register allocation

Given program as CPU instructions, assign physical registers to variables such that spilling overhead is minimized

Old problems in CS

Instruction scheduling

Given program as CPU instructions, reorder them to minimize pipeline stalls (without changing semantics)

Register allocation

Given program as CPU instructions, assign physical registers to variables such that spilling overhead is minimized

Problems are hard (NP-complete), intensively studied, and related

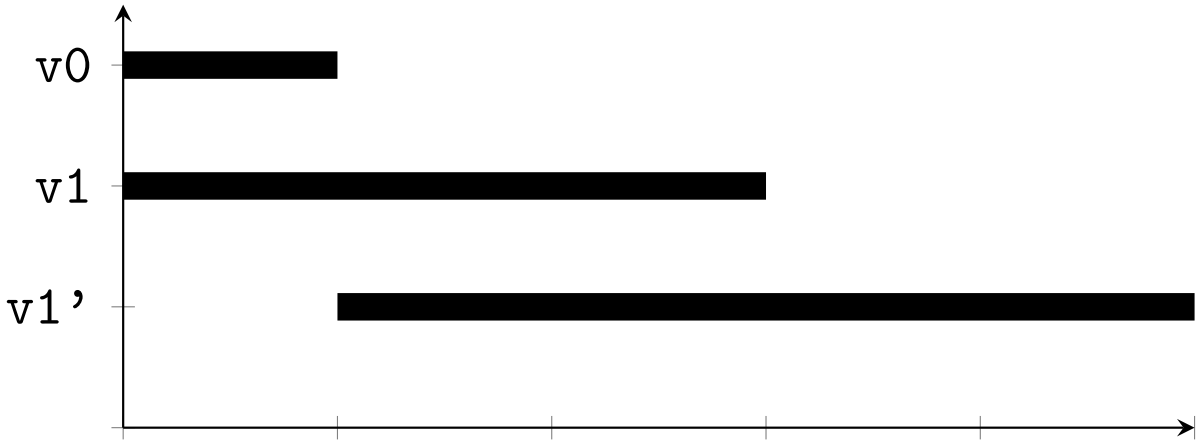
Classic approach A: Chaitin-Briggs

- Original idea in 1981 [CAC+81, Cha82], many improvements later

Classic approach A: Chaitin-Briggs

- Original idea in 1981 [CAC+81, Cha82], many improvements later
- Write program in *SSA form* to allocate *live ranges*

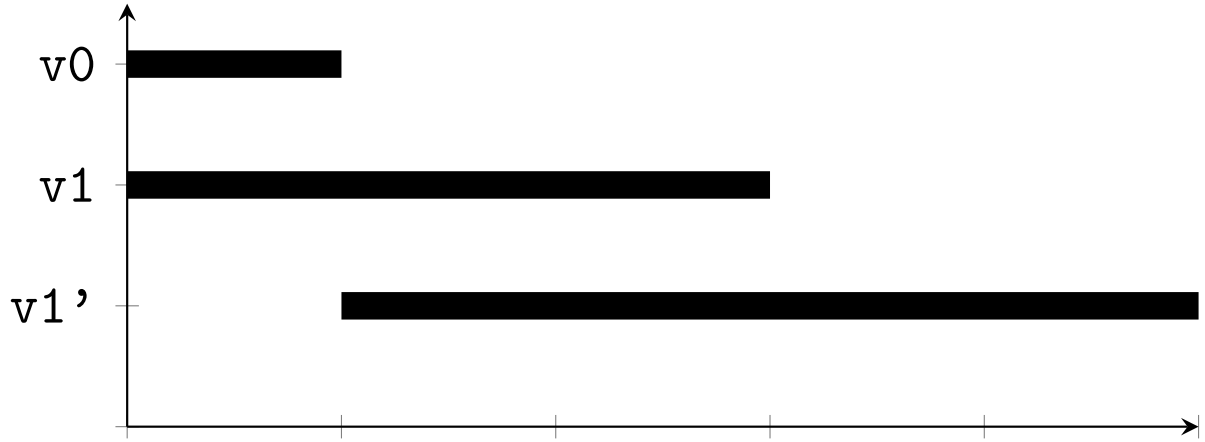
  `v1 = v1 + v0` $\mapsto$ `v1' = v1 + v0`



Classic approach A: Chaitin-Briggs

- Original idea in 1981 [CAC+81, Cha82], many improvements later
- Write program in *SSA form* to allocate *live ranges*

  `v1 = v1 + v0` $\mapsto$ `v1' = v1 + v0`

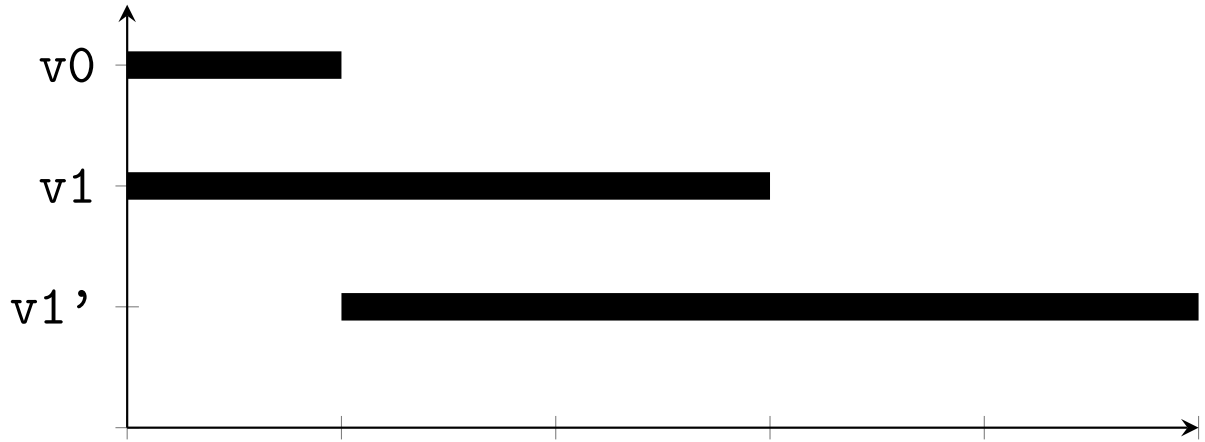


- Build interference graph G
  - Nodes represent live ranges
  - Edges represent interference between live ranges

Classic approach A: Chaitin-Briggs

- Original idea in 1981 [CAC+81, Cha82], many improvements later
- Write program in *SSA form* to allocate *live ranges*

  `v1 = v1 + v0` $\mapsto$ `v1' = v1 + v0`



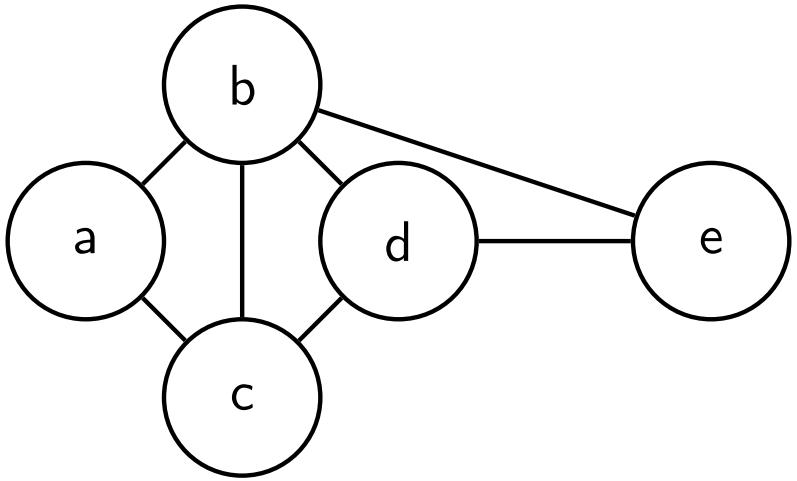
- Build interference graph G
  - Nodes represent live ranges
  - Edges represent interference between live ranges
- n-coloring exists if highest degree $< n$

Classic approach A: Chaitin-Briggs

while *G has no n-coloring* **do**
  while $\exists v \in G$ *with* $\deg(v) < n$ **do**
    Remove v and its edges from G and push v on stack;
  end
  if $G = \emptyset$ **then**
    while *Stack* $\neq []$ **do**
      Pop v from stack, add v back to G;
      Color v;
    end
  else
    Choose a node v to spill;
    Remove v and its edges from G;
  end
end

Classic approach A: Chaitin-Briggs

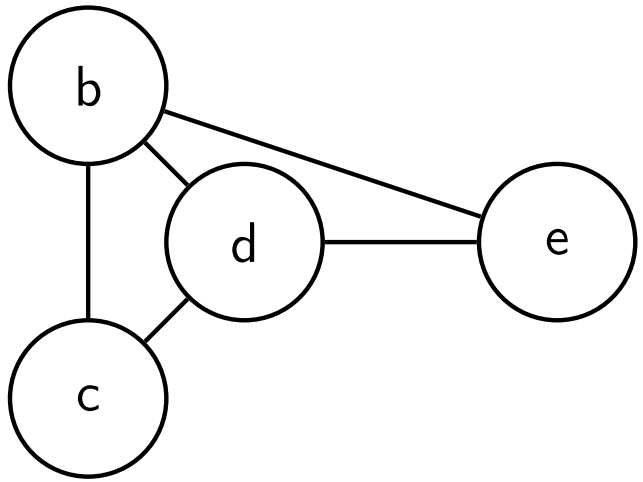
For example, $n = 3$.



Stack = []

Classic approach A: Chaitin-Briggs

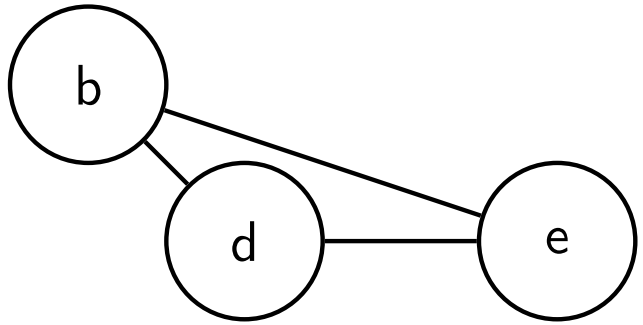
For example, $n = 3$.



Stack = $[a]$

Classic approach A: Chaitin-Briggs

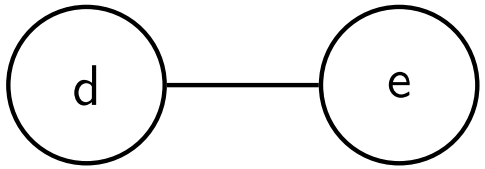
For example, $n = 3$.



Stack $= [a, c]$

Classic approach A: Chaitin-Briggs

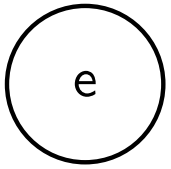
For example, $n = 3$.



Stack $= [a, c, b]$

Classic approach A: Chaitin-Briggs

For example, $n = 3$.



Stack $= [a, c, b, d]$

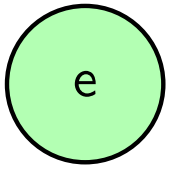
Classic approach A: Chaitin-Briggs

For example, $n = 3$.

Stack $= [a, c, b, d, e]$

Classic approach A: Chaitin-Briggs

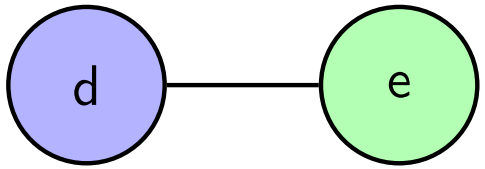
For example, $n = 3$.



Stack $= [a, c, b, d]$

Classic approach A: Chaitin-Briggs

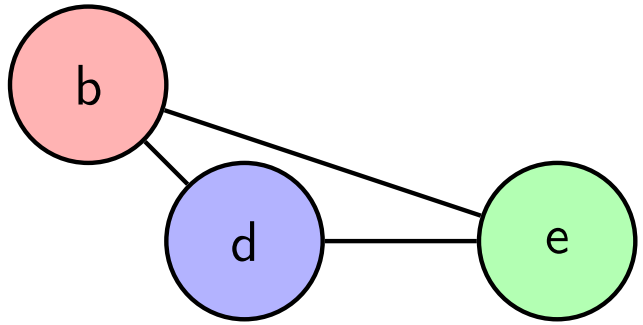
For example, $n = 3$.



Stack $= [a, c, b]$

Classic approach A: Chaitin-Briggs

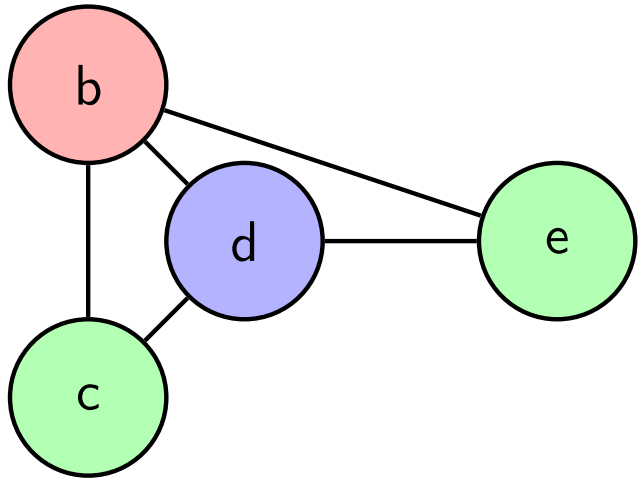
For example, $n = 3$.



Stack $= [a, c]$

Classic approach A: Chaitin-Briggs

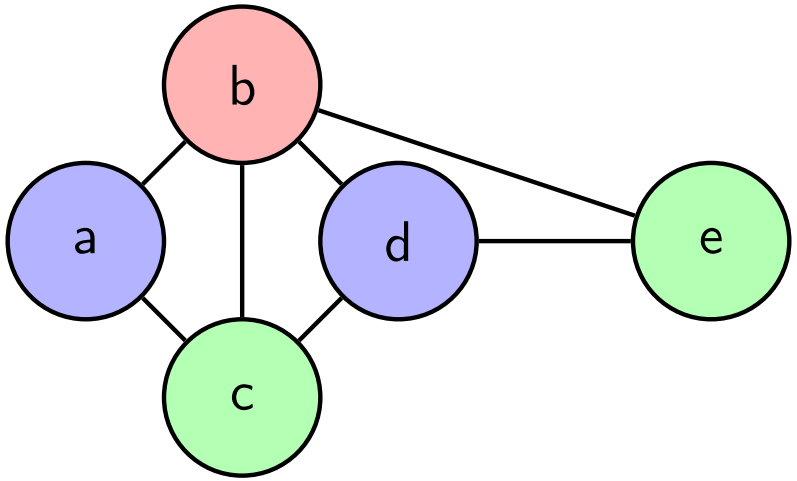
For example, $n = 3$.



Stack = $[a]$

Classic approach A: Chaitin-Briggs

For example, $n = 3$.



Stack = []

Classic approach A: Chaitin-Briggs

- Double representation of graph

Classic approach A: Chaitin-Briggs

- Double representation of graph
- How to choose node to spill?

Classic approach A: Chaitin-Briggs

- Double representation of graph
- How to choose node to spill?
- How to choose color to use?

Classic approach A: Chaitin-Briggs

- Double representation of graph
- How to choose node to spill?
- How to choose color to use?
- Many improvements
  - Rematerialization
  - Live range splitting

Classic approach A: Chaitin-Briggs

- Double representation of graph
- How to choose node to spill?
- How to choose color to use?
- Many improvements
  - Rematerialization
  - Live range splitting
- But still:
  - Multiple passes through program
  - Graph often rebuilt

Classic approach B: linear scan

- Chaitin-Briggs too slow for JIT

Classic approach B: linear scan

- Chaitin-Briggs too slow for JIT
- Linear scan by Poletto and Sarkar [PS99]

Classic approach B: linear scan

- Chaitin-Briggs too slow for JIT
- Linear scan by Poletto and Sarkar [PS99]

Classic approach B: linear scan

- Chaitin-Briggs too slow for JIT
- Linear scan by Poletto and Sarkar [PS99]

L = list of live ranges, sorted by start point;
A = list of allocated active live ranges;
foreach $l \in L$ **do**
    Remove expired live ranges from A, if any;
    if $length(A) = n$ **then**
        Choose live range $l' \in A$ that ends furthest away;
        Spill l', remove l' from A;
    end
    Allocate l, add l to A;
end

Classic approach B: linear scan

- Chaitin-Briggs too slow for JIT
- Linear scan by Poletto and Sarkar [PS99]

L = list of live ranges, sorted by start point;
A = list of allocated active live ranges;
foreach $l \in L$ **do**
    Remove expired live ranges from A, if any;
    if $length(A) = n$ **then**
        Choose live range $l' \in A$ that ends furthest away;
        Spill l', remove l' from A;
    end
    Allocate l, add l to A;
end

- Generated code only $\approx 10\%$ slower

Compilers: GCC

- Instruction scheduling, register allocation, instruction scheduling

Compilers: GCC

- Instruction scheduling, register allocation, instruction scheduling
- First integrated register allocator (IRA), then local (LRA)

Compilers: GCC

- Instruction scheduling, register allocation, instruction scheduling
- First integrated register allocator (IRA), then local (LRA)
- Like Chaitin-Briggs

Compilers: GCC

- Instruction scheduling, register allocation, instruction scheduling
- First integrated register allocator (IRA), then local (LRA)
- Like Chaitin-Briggs
- Region-based

Compilers: GCC

- Instruction scheduling, register allocation, instruction scheduling
- First integrated register allocator (IRA), then local (LRA)
- Like Chaitin-Briggs
- Region-based
- Region choice based on register pressure

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator
- Chaitin-Briggs assumes constant live ranges, machine code cannot change while running

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator
- Chaitin-Briggs assumes constant live ranges, machine code cannot change while running
- Based on linear-scan

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator
- Chaitin-Briggs assumes constant live ranges, machine code cannot change while running
- Based on linear-scan
- Priority queue with spill weights

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator
- Chaitin-Briggs assumes constant live ranges, machine code cannot change while running
- Based on linear-scan
- Priority queue with spill weights
- Live range splitting

Compilers: Clang

- Since LLVM 3.0, basic and greedy (and PBQP) allocator
- Chaitin-Briggs assumes constant live ranges, machine code cannot change while running
- Based on linear-scan
- Priority queue with spill weights
- Live range splitting
- Accomodates architecture-specific preferences
  Thumb-2: 16-bit encoding when using `r0`–`r7`

Compilers: ARM Compiler

- Commercial, closed-source until 5.x

Compilers: ARM Compiler

- Commercial, closed-source until 5.x
- Based on LLVM/Clang since 6.0 (2014)

Compilers: ARM Compiler

- Commercial, closed-source until 5.x
- Based on LLVM/Clang since 6.0 (2014)
- Use 5.06 (June 2016) for comparison

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)
- Most arithmetic instructions 1 cycle
  `eor r2, r0, r1, ror #24`

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)
- Most arithmetic instructions 1 cycle
  `eor r2, r0, r1, ror #24`
- Simple store to memory 1 cycle

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)
- Most arithmetic instructions 1 cycle
  `eor r2, r0, r1, ror #24`
- Simple store to memory 1 cycle
- Loads from memory $\geq$ 2 cycles

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)
- Most arithmetic instructions 1 cycle
  `eor r2, r0, r1, ror #24`
- Simple store to memory 1 cycle
- Loads from memory $\geq$ 2 cycles
- 3-stage pipeline

Case study: AES on Cortex-M3/M4

- 16 32-bit registers, 3 taken for `pc`, `sp`, (`lr`)
- Most arithmetic instructions 1 cycle
  `eor r2, r0, r1, ror #24`
- Simple store to memory 1 cycle
- Loads from memory ≥ 2 cycles
- 3-stage pipeline
- n loads can be pipelined to take $n + 1$ cycles

Case study: AES on Cortex-M3/M4

- Table-based, bitsliced, and masked bitsliced

Case study: AES on Cortex-M3/M4

- Table-based, bitsliced, and masked bitsliced
- Bitsliced S-box 113 gates [BP10], in SSA

Case study: AES on Cortex-M3/M4

- Table-based, bitsliced, and masked bitsliced
- Bitsliced S-box 113 gates [BP10], in SSA

Case study: AES on Cortex-M3/M4

- Table-based, bitsliced, and masked bitsliced
- Bitsliced S-box 113 gates [BP10], in SSA

```
y14 =  U3 +  U5
y13 =  U0 +  U6
 y9 =  U0 +  U3
 y8 =  U0 +  U5
 t0 =  U1 +  U2
 y1 =  t0 +  U7
 y4 =  y1 +  U3
y12 = y13 + y14
 y2 =  y1 +  U0
 y5 =  y1 +  U6
 y3 =  y5 +  y8
 t1 =  U4 + y12
y15 =  t1 +  U5
y20 =  t1 +  U1
 y6 = y15 +  U7
y10 = y15 +  t0
```

```
y11 = y20 +  y9
 y7 =  U7 + y11
y17 = y10 + y11
y19 = y10 +  y8
y16 =  t0 + y11
y21 = y13 + y16
y18 =  U0 + y16
 t2 = y12 x y15
 t3 =  y3 x  y6
 t4 =  t3 +  t2
 t5 =  y4 x  U7
 t6 =  t5 +  t2
 t7 = y13 x y16
 t8 =  y5 x  y1
 t9 =  t8 +  t7
t10 =  y2 x  y7
```

```
t11 = t10 +  t7
t12 =  y9 x y11
t13 = y14 x y17
t14 = t13 + t12
t15 =  y8 x y10
t16 = t15 + t12
t17 =  t4 + y20
t18 =  t6 + t16
t19 =  t9 + t14
t20 = t11 + t16
t21 = t17 + t14
t22 = t18 + y19
t23 = t19 + y21
t24 = t20 + y18
t25 = t21 + t22
(...)
```


iCIS | Digital Security
Radboud University

Why compilers are not ideal

- Compilers aim to produce fast binaries on average

Why compilers are not ideal

- Compilers aim to produce fast binaries on average
- Compilers aim to run reasonably fast on large code bases

Why compilers are not ideal

- Compilers aim to produce fast binaries on average
- Compilers aim to run reasonably fast on large code bases
- Compilers only do one attempt

Why compilers are not ideal

- Compilers aim to produce fast binaries on average
- Compilers aim to run reasonably fast on large code bases
- Compilers only do one attempt
- Compilers are complicated

Why compilers are not ideal

- Compilers aim to produce fast binaries on average
- Compilers aim to run reasonably fast on large code bases
- Compilers only do one attempt
- Compilers are complicated
- (Also, qhasm, but requires manual spill code generation)

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side
- Then allocate greedily, keep output in registers

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side
- Then allocate greedily, keep output in registers
- If registers are full
  - Free register with expired variable
  - Otherwise, free register with longest distance until reuse

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side
- Then allocate greedily, keep output in registers
- If registers are full
  - Free register with expired variable
  - Otherwise, free register with longest distance until reuse
- Detect direct recomputation, can be cheaper than loading from memory

Our scheduler and register allocator

- Focus only on ARM's three-operand instructions
- Multiple strategies implemented, designed to 'play round'
- Nondeterministic due to hash randomization
- First reschedule, decrease the length of live ranges
  - Push down based on left-hand side
  - Push up based on right-hand side
- Then allocate greedily, keep output in registers
- If registers are full
  - Free register with expired variable
  - Otherwise, free register with longest distance until reuse
- Detect direct recomputation, can be cheaper than loading from memory
- Source code in public domain:
  `https://github.com/Ko-/aes-armcortexm`

iCIS | Digital Security
Radboud University

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]
- Results for 113-instruction S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]
- Results for 113-instruction S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

- Most recent compiler versions, 'best' flags

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]
- Results for 113-instruction S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

- Most recent compiler versions, 'best' flags
- Other compilers also insert arithmetic and move instructions

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]
- Results for 113-instruction S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

- Most recent compiler versions, 'best' flags
- Other compilers also insert arithmetic and move instructions
- Results for 454-instruction masked S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	330	185	332	135
Stores	126	145	132	99

Results

- Used in fastest AES implementations for Cortex-M3/M4 [SS16]
- Results for 113-instruction S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

- Most recent compiler versions, 'best' flags
- Other compilers also insert arithmetic and move instructions
- Results for 454-instruction masked S-box

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	330	185	332	135
Stores	126	145	132	99

- (Excluding 32 loads for randomness)

Results

Algorithm	Speed (cycles)		ROM (bytes)		RAM (bytes)	
	M3	M4	Code	Data	I/O	Stack
AES-128-CTR	546.3	**554.4**	2192	1024	192 $+2m$	72
Bitsliced AES-128-CTR	1616.6	**1617.6**	12120	12	368 $+2m$	108
Masked bitsliced AES-128-CTR	N/A	**7422.6**	39916	12	368 $+2m$	1588
AES-128 KS	289.8	294.8	902	1024	176	32
Bitsliced AES-128 KS	1027.8	1033.8	3434	1036	368	188
Masked bitsliced AES-128 KS	1027.8	1033.8	3434	1036	368	188

More on full AES in [SS16]

Thanks. . .

. . . for your attention!

Paper and code at
https://ko.stoffelen.nl/

iCIS | Digital Security
Radboud University

References I

Joan Boyar and René Peralta.
A new combinational logic minimization technique with applications to cryptology.
In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010.
http://eprint.iacr.org/2009/191/.

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.

G. J. Chaitin.
Register allocation & spilling via graph coloring.
In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105. ACM, 1982.

Massimiliano Poletto and Vivek Sarkar.
Linear scan register allocation.
ACM Trans. Program. Lang. Syst., 21(5):895–913, September 1999.

References II

Peter Schwabe and Ko Stoffelen.
All the AES you need on Cortex-M3 and M4.
In *Selected Areas in Cryptography – SAC 2016*, LNCS. Springer, 2016.
https://eprint.iacr.org/2016/714/.