Instruction Scheduling and Register Allocation on ARM Cortex-M

Ko Stoffelen
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

\[
v_1 = v_1 + v_0 \quad \rightarrow \quad v_1' = v_1 + v_0
\]

![Graph showing live ranges v0, v1, v1']
Classic approach A: Chaitin-Briggs

- Original idea in 1981 [CAC+81, Cha82], many improvements later
- Write program in SSA form to allocate live ranges
  \[ v_1 = v_1 + v_0 \implies v_1' = v_1 + v_0 \]

- 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
  \[ v_1 = v_1 + v_0 \implies v_1' = v_1 + v_0 \]

- 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]

$L = \text{list of live ranges, sorted by start point};
A = \text{list of allocated active live ranges};$

\begin{verbatim}
foreach $l \in L$ do
    Remove expired live ranges from $A$, if any;
    if $\text{length}(A) = n$ 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
\end{verbatim}
Classic approach B: linear scan

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

$L = \text{list of live ranges, sorted by start point};$
$A = \text{list of allocated active live ranges};$

\textbf{foreach} \ l \in L \ \textbf{do}

- Remove expired live ranges from $A$, if any;
- \textbf{if} length($A$) = $n$ \textbf{then}
  - Choose live range $l' \in A$ that ends furthest away;
  - Spill $l'$, remove $l'$ from $A$;
\textbf{end}

- Allocate $l$, add $l$ to $A$;
\textbf{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 ≥ 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 ≥ 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

\[
\begin{align*}
y_{14} &= U_3 + U_5 \\
y_{13} &= U_0 + U_6 \\
y_9 &= U_0 + U_3 \\
y_8 &= U_0 + U_5 \\
t_0 &= U_1 + U_2 \\
y_1 &= t_0 + U_7 \\
y_4 &= y_1 + U_3 \\
y_{12} &= y_{13} + y_{14} \\
y_2 &= y_1 + U_0 \\
y_5 &= y_1 + U_6 \\
y_3 &= y_5 + y_8 \\
t_1 &= U_4 + y_{12} \\
y_{15} &= t_1 + U_5 \\
y_{20} &= t_1 + U_1 \\
y_6 &= y_{15} + U_7 \\
y_{10} &= y_{15} + t_0
\end{align*}
\]

\[
\begin{align*}
y_{11} &= y_{20} + y_9 \\
y_7 &= U_7 + y_{11} \\
y_{17} &= y_{10} + y_{11} \\
y_{19} &= y_{10} + y_8 \\
y_{16} &= t_0 + y_{11} \\
y_{21} &= y_{13} + y_{16} \\
y_{18} &= U_0 + y_{16} \\
y_{12} &= y_{13} + y_{14} \\
y_3 &= y_3 + y_6 \\
y_4 &= t_3 + t_2 \\
t_5 &= y_4 + U_7 \\
t_6 &= t_5 + t_2 \\
t_7 &= y_{13} + y_{16} \\
t_8 &= y_5 + y_1 \\
t_9 &= t_8 + t_7 \\
t_{10} &= y_2 + y_7 \\
t_{11} &= y_{10} + t_0 \\
t_{12} &= y_8 + y_{10} \\
t_{13} &= y_{14} + y_{17} \\
t_{14} &= t_{13} + t_{12} \\
t_{15} &= y_8 + y_{10} \\
t_{16} &= t_{15} + t_{12} \\
t_{17} &= t_4 + y_{20} \\
t_{18} &= t_6 + t_{16} \\
t_{19} &= t_9 + t_{14} \\
t_{20} &= t_{11} + t_{16} \\
t_{21} &= t_{17} + t_{14} \\
t_{22} &= t_{18} + y_{19} \\
t_{23} &= t_{19} + y_{21} \\
t_{24} &= t_{20} + y_{18} \\
t_{25} &= t_{21} + t_{22} \\
(\ldots) 
\end{align*}
\]
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

- 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
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

Source code in public domain: https://github.com/Ko-/aes-armcortexm
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

Source code in public domain: https://github.com/Ko-/aes-armcortexm
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
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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Stores</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

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

Results for 454-instruction masked S-box

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>330</td>
<td>185</td>
<td>332</td>
<td>135</td>
</tr>
<tr>
<td>Stores</td>
<td>126</td>
<td>145</td>
<td>132</td>
<td>99</td>
</tr>
</tbody>
</table>

• (Excluding 32 loads for randomness)
Results

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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Stores</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

- Most recent compiler versions, ‘best’ flags
Results

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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Stores</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Stores</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>330</td>
<td>185</td>
<td>332</td>
<td>135</td>
</tr>
<tr>
<td>Stores</td>
<td>126</td>
<td>145</td>
<td>132</td>
<td>99</td>
</tr>
</tbody>
</table>
Results

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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>46</td>
<td>32</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Stores</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Compilers</th>
<th>GCC</th>
<th>Clang</th>
<th>ARM Compiler</th>
<th>Our tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>330</td>
<td>185</td>
<td>332</td>
<td>135</td>
</tr>
<tr>
<td>Stores</td>
<td>126</td>
<td>145</td>
<td>132</td>
<td>99</td>
</tr>
</tbody>
</table>

- (Excluding 32 loads for randomness)
## Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed (cycles)</th>
<th>ROM (bytes)</th>
<th>RAM (bytes)</th>
<th>I/O</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M3</td>
<td>M4</td>
<td>Code</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>AES-128-CTR</td>
<td>546.3</td>
<td>554.4</td>
<td>2192</td>
<td>1024</td>
<td>192</td>
</tr>
<tr>
<td>Bitsliced AES-128-CTR</td>
<td>1616.6</td>
<td>1617.6</td>
<td>12120</td>
<td>12</td>
<td>368</td>
</tr>
<tr>
<td>Masked bitsliced AES-128-CTR</td>
<td>N/A</td>
<td>7422.6</td>
<td>39916</td>
<td>12</td>
<td>368</td>
</tr>
<tr>
<td>AES-128 KS</td>
<td>289.8</td>
<td>294.8</td>
<td>902</td>
<td>1024</td>
<td>176</td>
</tr>
<tr>
<td>Bitsliced AES-128 KS</td>
<td>1027.8</td>
<td>1033.8</td>
<td>3434</td>
<td>1036</td>
<td>368</td>
</tr>
<tr>
<td>Masked bitsliced AES-128 KS</td>
<td>1027.8</td>
<td>1033.8</td>
<td>3434</td>
<td>1036</td>
<td>368</td>
</tr>
</tbody>
</table>

More on full AES in [SS16]
Thanks...

...for your attention!

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

Joan Boyar and René Peralta.
A new combinational logic minimization technique with applications to cryptology.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E.
Hopkins, and Peter W. Markstein.
Register allocation via coloring.

G. J. Chaitin.
Register allocation & spilling via graph coloring.

Massimiliano Poletto and Vivek Sarkar.
Linear scan register allocation.
Peter Schwabe and Ko Stoffelen.
All the AES you need on Cortex-M3 and M4.
https://eprint.iacr.org/2016/714/.