Optimizing small AArch64 cores

Stories from the trenches

- Story of embedded team optimizing Cortex-A320

- Overall theme of AArch64 backend pattern

What did we do?

- What to optimize
  - Coremark
  - Compare hand-written assembly kernels (CMSIS-DSP library)

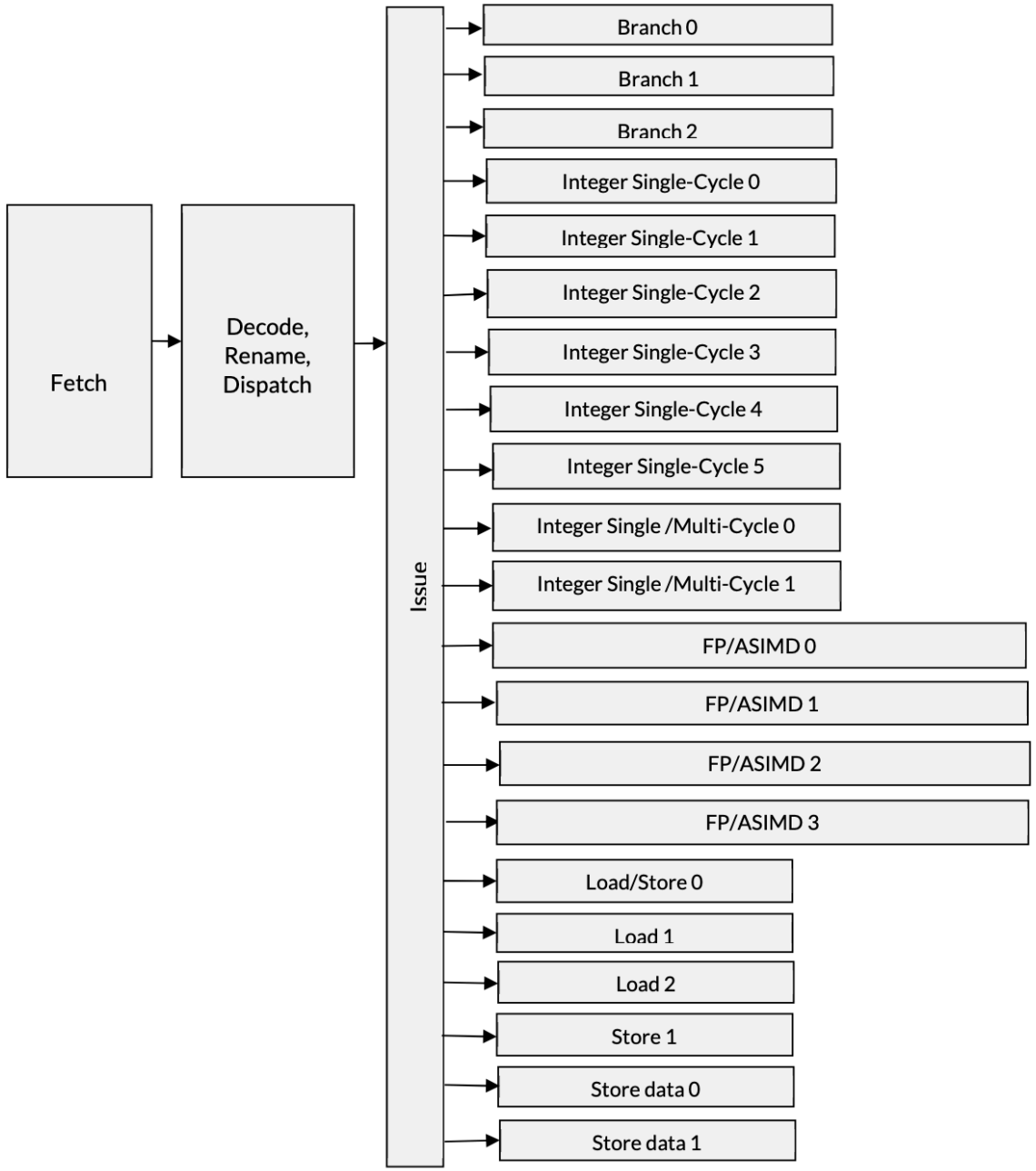
background

- Small in-order cores/big Out of Order cores

- Cortex-A320 -> more than 40% smaller than Cortex-A520
  - Wearables, IoT, smart devices
  - By themselves, in clusters
  - Armv9.2-A features

Cortex-A320 overview (from Software Optimization Guide)

Figure 2-1 Arm® Cortex-X4 Core pipeline



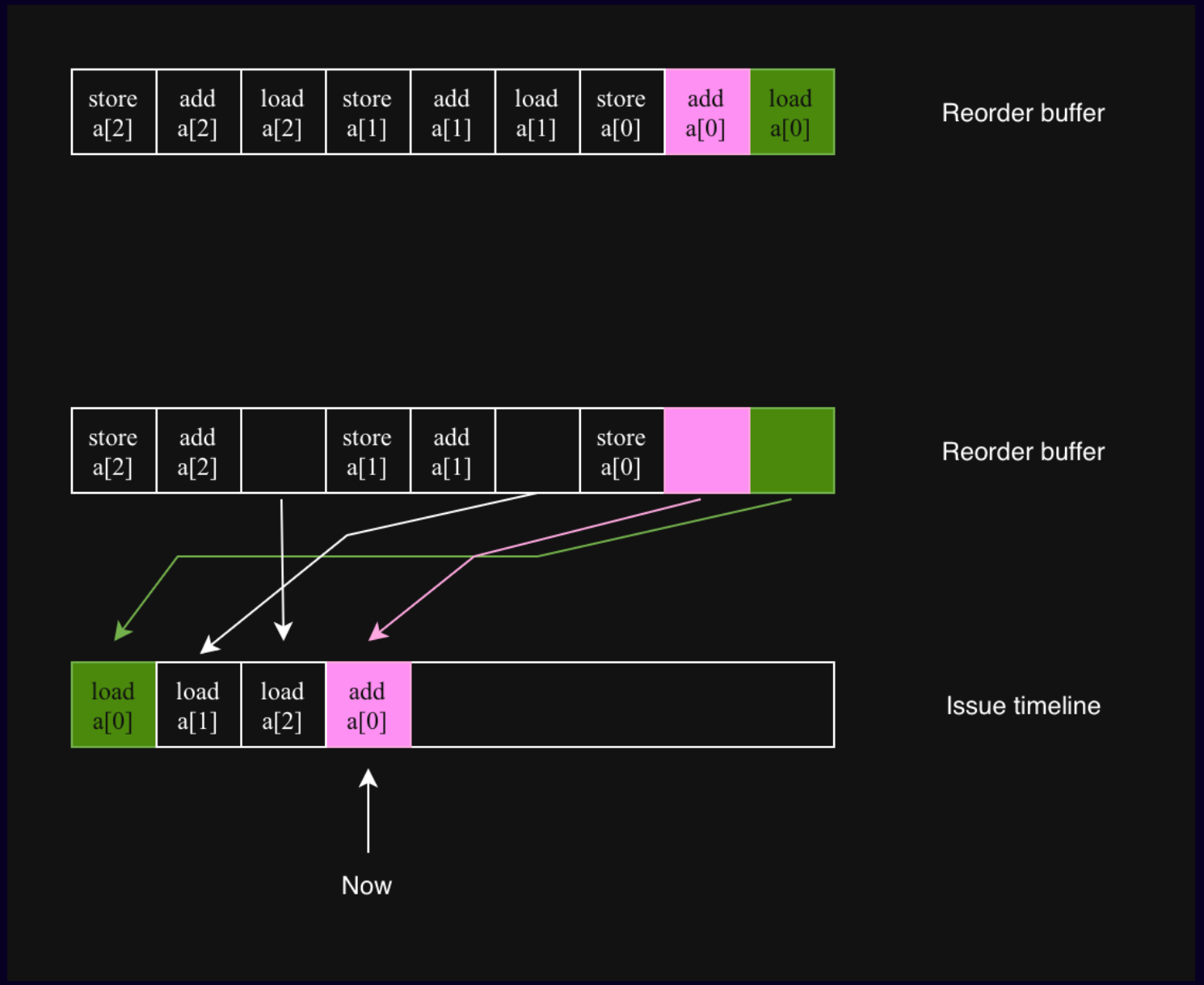
Cortex-A320:

- In-order
- Single issue

Cortex-X4:

- Out of Order
- 10 ops issue width
- Register renaming
- Reorder buffer (ROB)

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```



So..

- No runtime scheduling
- Do at compile time
- Bit topsy-turvy for someone coming from embedded

Optimizations we’re working on

- Loop unrolling
- Software pipelining
- Load/Store post increment

Loop unrolling

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```

```
// x0 = a, x2 = byte offset
.loop:
    ldr  w4, [x0, x2]        // w4 = a[i]
    add  w4, w4, #1          // w4 = a[i] + 1
    str  w4, [x0, x2]        // a[i] = w4
    add  x2, x2, #4          // i++
    cmp  x2, x3              // i == n?
    b.ne .loop
```

- More instructions to schedule, so more ways to avoid stalls
- Branch overhead

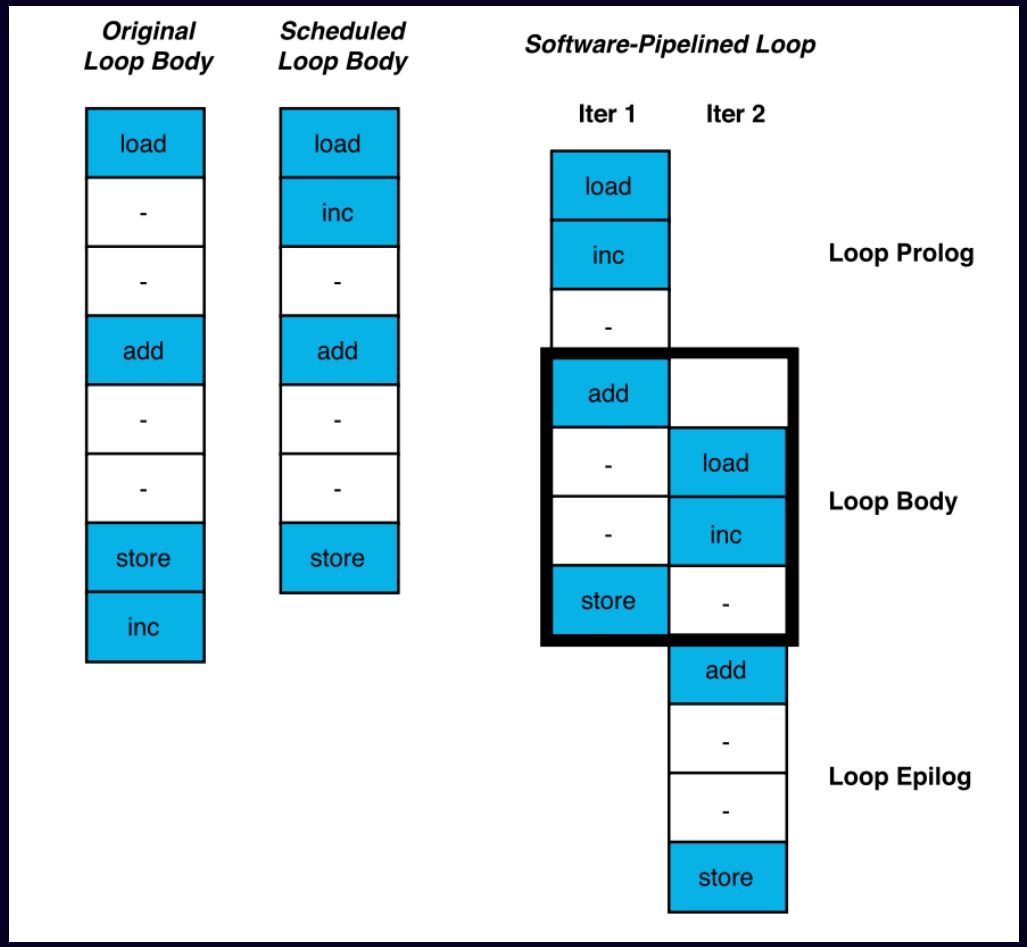
Loop unrolling

```
.loop:
    ldr  w3, [x0]           // a[i+0]
    ldr  w4, [x0, #4]       // a[i+1]
    ldr  w5, [x0, #8]       // a[i+2]
    ldr  w6, [x0, #12]      // a[i+3]
    add  w3, w3, #1
    add  w4, w4, #1
    add  w5, w5, #1
    add  w6, w6, #1
    str  w3, [x0], #4
    str  w4, [x0], #4
    str  w5, [x0], #4
    str  w6, [x0], #4
    cmp  x0, x2
    b.ne .loop
```

- Introduced `-mllvm -aarch64-force-unroll-threshold=<x>`
- 10% Coremark speedup

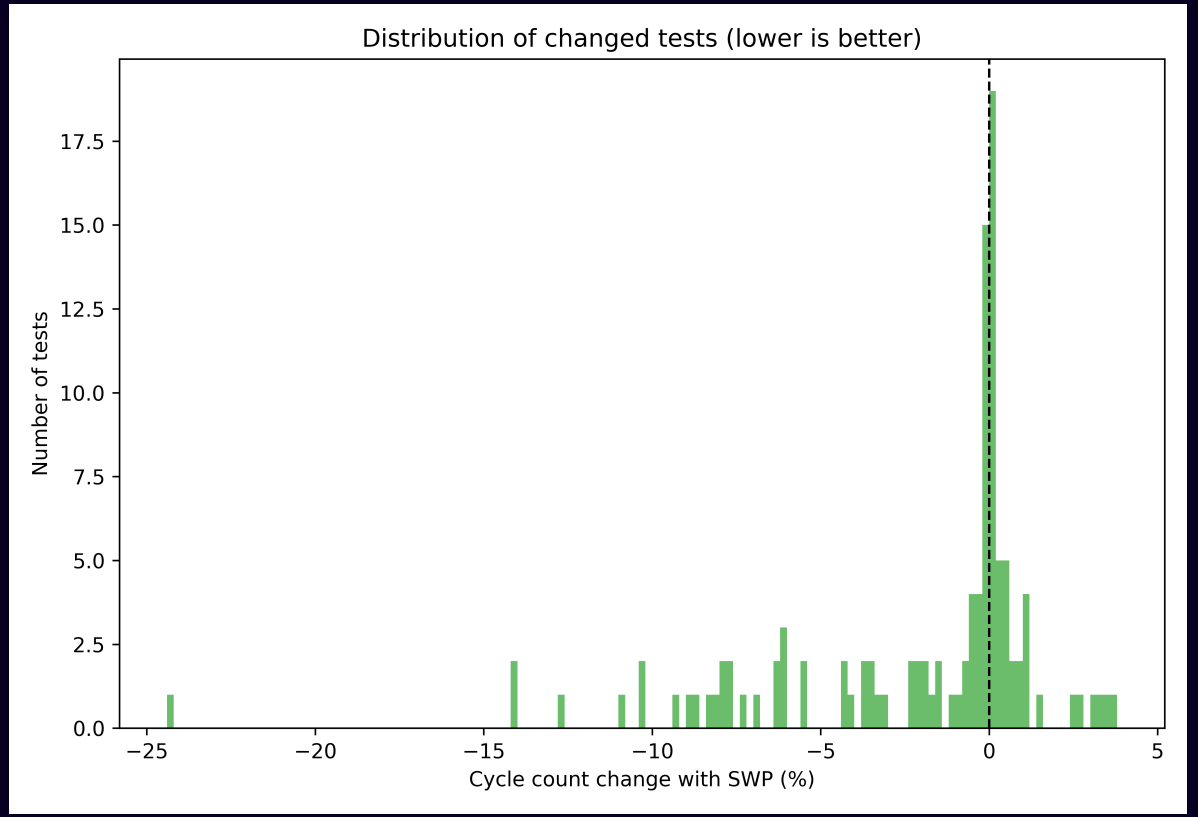
Software pipelining

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```



Software pipelining

- Pass in LLVM: MachinePipeliner
- From 2015
- Software Pipelining for AArch64 by Fujitsu in 2024
- We are adding register pressure calculation
- Lots of improvements on CMSIS-DSP
- Regressions need looking at



Load/Store post increment

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```

->

```
while (byte_offset != byte_limit) {
    int *p = (int *)((char *)a + byte_offset);
    *p = *p + 1;
    byte_offset += 4;
}
```

```
// x0 = a, x2 = byte offset, x3 = byte limit
.loop:
    ldr  w4, [x0, x2]       // load *p
    add  w4, w4, #1         // *p + 1
    str  w4, [x0, x2]       // store *p
    add  x2, x2, #4         // byte_offset += 4
    cmp  x2, x3             // byte_offset == byte_limit?
    b.ne .loop
```

Load/Store post increment

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```

->

```
while (p != end) {
    *p = *p + 1;
    p++;
}
```

```
// x0 = p, x2 = end
.loop:
    ldr  w3, [x0]          // load *p
    add  w3, w3, #1        // *p + 1
    str  w3, [x0]          // store *p
    add  x0, x0, #4        // p++
    cmp  x0, x2            // p == end?
    b.ne .loop
```

Load/Store post increment

```
for(int i=0; i < n; i++)
    a[i] = a[i] + 1;
```

->

```
while (p != end) {
    *p = *p + 1;
    p++;
}
```

```
// x0 = p, x2 = end
.loop:
    ldr  w3, [x0]          // load *p
    add  w3, w3, #1        // *p + 1
    str  w3, [x0], #4      // store *p, p++
    cmp  x0, x2            // p == end?
    b.ne .loop
```


Load/Store post increment

- Load/store followed by add to the base
- Forming post-increment
  - Isel: load/store is only use of address in block
  - AArch64LoadStoreOptimizer: more than one use

Loop Strength Reduce

- LSR transforms loop induction variables to reduce their cost
- No "post increment by vector length" instructions in SVE
- TargetTransformInfo::getPreferredAddressingMode
  - None, PreIndexed, PostIndexed
  - Affects generated patterns, pattern pruning, cost calculation
  - Introduced All, to consider pre- and post-indexed and offset addressing
  - Still a lot to do to mitigate regressions

Loop Strength Reduce

Patches merged:

- [LSR] Remove unnecessary WidestFixupType (NFC) (#185013)
- [LSR] Make OptimizeLoopTermCond able to handle some non-cmp conditions (#165590)
- [LSR] Insert the transformed IV increment in the user block (#169515)
- [LSR] Don't count conditional loads/store as enabling pre/post-index (#159573)
- [LSR] Add an addressing mode that considers all addressing modes (#158110)
- [LSR] Account for hardware loop instructions (#147958)
- [LSR] Make canHoistIVInc allow non-integer types (#143707)

More to do:

- Uses outside of loop handled badly (patch in upstream review).
- Discount to AddRecCost due to postincrement applied even when there's no load/store to postincrement.
- SVE vscale counted as occupying a register in cases where it's folded into addressing mode.

Wrap up

- Overall a lot of improvements for Cortex-A320 style CPUs
- Next: vectorization, partner feedback

- Optimization suggestions, collaborations:
  - please come and talk to us

- me -> Ties Stuij -> @stuij
- Load/Store post increment -> John Brawn -> @john-brawn-arm
- Software pipelining -> William Huynh -> @saturn691

- Embedded Toolchain for Arm -> https://github.com/arm/arm-toolchain
- Think of us embedded folks

Attribution

- Software pipelining example diagram from “Basic Instruction Scheduling (and Software Pipelining)”