Fast and Clean: Auditable high-performance assembly via constraint solving

Amin Abdulrahman, Hanno Becker, Matthias J. Kannwischer, Fabien Klein

`https://github.com/slothy-optimizer/slothy`

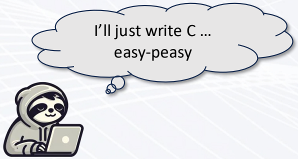
6 September 2024, Conference on Cryptographic Hardware and Embedded Systems, Halifax, Canada

Motivation

Good cryptographic engineering is **HARD**

Balancing act between ...

- Simplicity



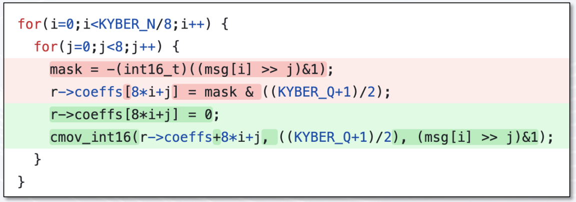
```c
void ntt(int16_t r[256]) {
  unsigned int len, start, j, k;
  int16_t t, zeta;

  k = 1;
  for(len = 128; len >= 2; len >>= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas[k++];
      for(j = start; j < start + len; j++) {
        t = fqmul(zeta, r[j + len]);
        r[j + len] = r[j] - t;
        r[j] = r[j] + t;
      }
    }
  }
}
```

Motivation

Good cryptographic engineering is **HARD**

Balancing act between ...
- Simplicity
- Security



```
for(i=0;i<KYBER_N/8;i++) {
  for(j=0;j<8;j++) {
    mask = -(int16_t)((msg[i] >> j)&1);
    r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);
    r->coeffs[8*i+j] = 0;
    cmov_int16(r->coeffs+8*i+j, ((KYBER_Q+1)/2), (msg[i] >> j)&1);
  }
}
```

Motivation

Good cryptographic engineering is **HARD**

Balancing act between ...

- Simplicity
- Security
- Performance

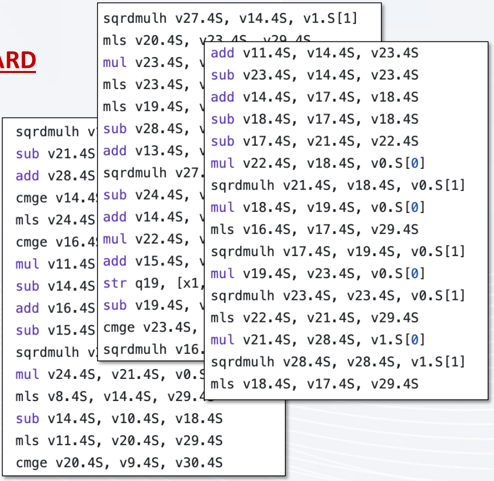
```
.macro mulmodq dst, src, const, idx0, idx1
        vqrdmulhq   t2,  \src, \const, \idx1
        vmulq       \dst,  \src, \const, \idx0
        vmlsq       \dst,  t2, consts, 0
.endm
```

Motivation

Good cryptographic engineering is **HARD**

Balancing act between …

- Simplicity
- Security
- Performance
- More performance



```
sqrdmulh v
sub v21.4S
add v28.4S
cmge v14.4
mls v24.4S
cmge v16.4
mul v11.4S
sub v14.4S
add v16.4S
sub v15.4S
sqrdmulh v
mul v24.4S, v21.4S, v0.S
mls v8.4S, v14.4S, v29.4
sub v14.4S, v10.4S, v18.4S
mls v11.4S, v20.4S, v29.4S
cmge v20.4S, v9.4S, v30.4S
```

```
sqrdmulh v27.4S, v14.4S, v1.S[1]
mls v20.4S, v23.4S, v29.4S
mul v23.4S, v
mls v23.4S, v
mls v19.4S, v
sub v28.4S, v
add v13.4S, v
sqrdmulh v27.
sub v24.4S, v
add v14.4S, v
mul v22.4S, v
add v15.4S, v
str q19, [x1,
sub v19.4S, v
cmge v23.4S,
sqrdmulh v16.
```

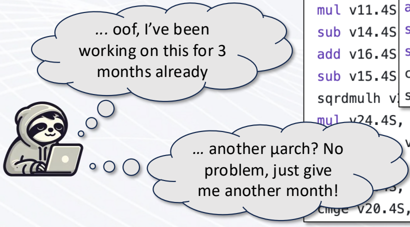
```
add v11.4S, v14.4S, v23.4S
sub v23.4S, v14.4S, v23.4S
add v14.4S, v17.4S, v18.4S
sub v18.4S, v17.4S, v18.4S
sub v17.4S, v21.4S, v22.4S
mul v22.4S, v18.4S, v0.S[0]
sqrdmulh v21.4S, v18.4S, v0.S[1]
mul v18.4S, v19.4S, v0.S[0]
mls v16.4S, v17.4S, v29.4S
sqrdmulh v17.4S, v19.4S, v0.S[1]
mul v19.4S, v23.4S, v0.S[0]
sqrdmulh v23.4S, v23.4S, v0.S[1]
mls v22.4S, v21.4S, v29.4S
mul v21.4S, v28.4S, v1.S[0]
sqrdmulh v28.4S, v28.4S, v1.S[1]
mls v18.4S, v17.4S, v29.4S
```

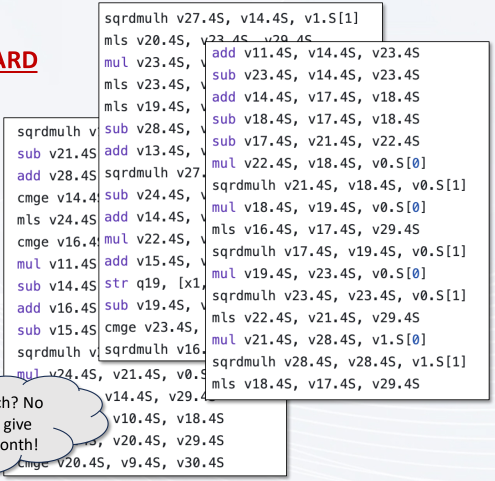
Motivation

Good cryptographic engineering is **HARD**

Balancing act between ...

- Simplicity
- Security
- Performance
- More performance
- Effort


... oof, I've been working on this for 3 months already
... another µarch? No problem, just give me another month!



```
sqrdmulh v
sub v21.4S
add v28.4S
cmge v14.4
mls v24.4S
cmge v16.4
mul v11.4S
sub v14.4S
add v16.4S
sub v15.4S
sqrdmulh v
mul v24.4S, v21.4S, v0.S
         v14.4S, v29.4
         v10.4S, v18.4S
       S, v20.4S, v29.4S
cmge v20.4S, v9.4S, v30.4S
```

```
sqrdmulh v27.4S, v14.4S, v1.S[1]
mls v20.4S, v23.4S, v29.4S
mul v23.4S, v
mls v23.4S, v
mls v19.4S, v
sub v28.4S, v
add v13.4S, v
sqrdmulh v27.
sub v24.4S, v
add v14.4S, v
mul v22.4S, v
add v15.4S, v
str q19, [x1,
sub v19.4S, v
cmge v23.4S,
sqrdmulh v16.
```

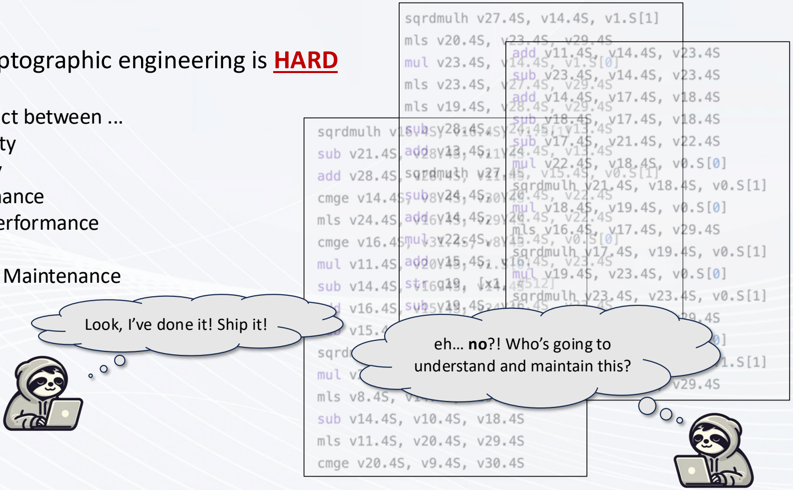
```
add v11.4S, v14.4S, v23.4S
sub v23.4S, v14.4S, v23.4S
add v14.4S, v17.4S, v18.4S
sub v18.4S, v17.4S, v18.4S
sub v17.4S, v21.4S, v22.4S
mul v22.4S, v18.4S, v0.S[0]
sqrdmulh v21.4S, v18.4S, v0.S[1]
mul v18.4S, v19.4S, v0.S[0]
mls v16.4S, v17.4S, v29.4S
sqrdmulh v17.4S, v19.4S, v0.S[1]
mul v19.4S, v23.4S, v0.S[0]
sqrdmulh v23.4S, v23.4S, v0.S[1]
mls v22.4S, v21.4S, v29.4S
mul v21.4S, v28.4S, v1.S[0]
sqrdmulh v28.4S, v28.4S, v1.S[1]
mls v18.4S, v17.4S, v29.4S
```

Motivation

Good cryptographic engineering is **HARD**

Balancing act between ...

- Simplicity
- Security
- Performance
- More performance
- Effort
- Audit & Maintenance


Look, I've done it! Ship it!
eh… **no**?! Who's going to understand and maintain this?

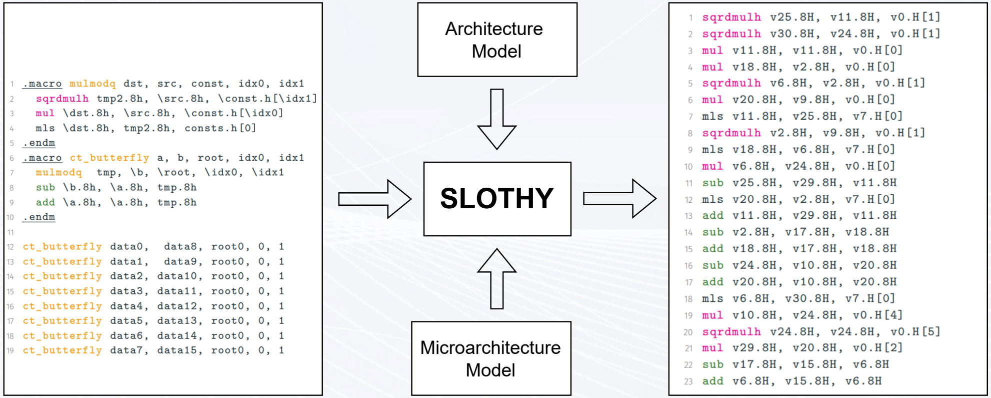
```
sqrdmulh v27.4S, v14.4S, v1.S[1]
mls v20.4S, v23.4S, v29.4S
mul v23.4S, v14.4S, v1.S[0]
mls v23.4S, v27.4S, v29.4S
mls v19.4S, v28.4S, v29.4S
```

```
add v11.4S, v14.4S, v23.4S
sub v23.4S, v14.4S, v23.4S
add v14.4S, v17.4S, v18.4S
sub v18.4S, v17.4S, v18.4S
sub v17.4S, v21.4S, v22.4S
mul v22.4S, v18.4S, v0.S[0]
sqrdmulh v21.4S, v18.4S, v0.S[1]
mul v18.4S, v19.4S, v0.S[0]
mls v16.4S, v17.4S, v29.4S
sqrdmulh v17.4S, v19.4S, v0.S[1]
mul v19.4S, v23.4S, v0.S[0]
sqrdmulh v23.4S, v23.4S, v0.S[1]
```

```
sqrdmulh v1
sub v21.4S,
add v28.4S,
cmge v14.4S,
mls v24.4S,
cmge v16.4S,
mul v11.4S,
sub v14.4S,
mls v8.4S,
sub v14.4S, v10.4S, v18.4S
mls v11.4S, v20.4S, v29.4S
cmge v20.4S, v9.4S, v30.4S
```

Motivation: What is SLOTHY?

Goal: Write μarch-independent code + automate μarch-specific changes



```
.macro mulmodq dst, src, const, idx0, idx1
  sqrdmulh tmp2.8h, \src.8h, \const.h[\idx1]
  mul \dst.8h, \src.8h, \const.h[\idx0]
  mls \dst.8h, tmp2.8h, consts.h[0]
.endm
.macro ct_butterfly a, b, root, idx0, idx1
  mulmodq  tmp, \b, \root, \idx0, \idx1
  sub \b.8h, \a.8h, tmp.8h
  add \a.8h, \a.8h, tmp.8h
.endm

ct_butterfly data0,  data8, root0, 0, 1
ct_butterfly data1,  data9, root0, 0, 1
ct_butterfly data2, data10, root0, 0, 1
ct_butterfly data3, data11, root0, 0, 1
ct_butterfly data4, data12, root0, 0, 1
ct_butterfly data5, data13, root0, 0, 1
ct_butterfly data6, data14, root0, 0, 1
ct_butterfly data7, data15, root0, 0, 1
```

```
sqrdmulh v25.8H, v11.8H, v0.H[1]
sqrdmulh v30.8H, v24.8H, v0.H[1]
mul v11.8H, v11.8H, v0.H[0]
mul v18.8H, v2.8H, v0.H[0]
sqrdmulh v6.8H, v2.8H, v0.H[1]
mul v20.8H, v9.8H, v0.H[0]
mls v11.8H, v25.8H, v7.H[0]
sqrdmulh v2.8H, v9.8H, v0.H[1]
mls v18.8H, v6.8H, v7.H[0]
mul v6.8H, v24.8H, v0.H[0]
sub v25.8H, v29.8H, v11.8H
mls v20.8H, v2.8H, v7.H[0]
add v11.8H, v29.8H, v11.8H
sub v2.8H, v17.8H, v18.8H
add v18.8H, v17.8H, v18.8H
sub v24.8H, v10.8H, v20.8H
add v20.8H, v10.8H, v20.8H
mls v6.8H, v30.8H, v7.H[0]
mul v10.8H, v24.8H, v0.H[4]
sqrdmulh v24.8H, v24.8H, v0.H[5]
mul v29.8H, v20.8H, v0.H[2]
sub v17.8H, v15.8H, v6.8H
add v6.8H, v15.8H, v6.8H
```

Motivation: Kyber NTT

```
.macro mulmodq dst, src, const, idx0, idx1
  sqrdmulh tmp2.8h, \src.8h, \const.h[\idx1]
  mul \dst.8h, \src.8h, \const.h[\idx0]
  mla \dst.8h, tmp2.8h, consts.h[0]
.endm
.macro ct_butterfly a, b, root, idx0, idx1
  mulmodq  tmp, \b, \root, \idx0, \idx1
  sub \b.8h, \a.8h, tmp.8h
  add \a.8h, \a.8h, tmp.8h
.endm

ct_butterfly data0,  data8, root0, 0, 1
ct_butterfly data1,  data9, root0, 0, 1
ct_butterfly data2, data10, root0, 0, 1
ct_butterfly data3, data11, root0, 0, 1
ct_butterfly data4, data12, root0, 0, 1
ct_butterfly data5, data13, root0, 0, 1
ct_butterfly data6, data14, root0, 0, 1
ct_butterfly data7, data15, root0, 0, 1
```

		cycles
	Cortex-A72	
[BHK+22]	handwritten	1200
Ours	clean	1307
Ours	SLOTHY	932
	Cortex-A55	
[BHK+22]	handwritten	1245
Ours	clean	1914
Ours	SLOTHY	891

Contributions

- SLOTHY: Register allocation, instruction scheduling, and software pipelining as a constraint satisfaction problem
- Implementation of SLOTHY using CP-SAT from Google's OR-Tools
- Architecture models: MVE and Neon (partial)
- μArchitecture models: Cortex-M55, Cortex-M85, Cortex-A55, Cortex-A72
    - By now: Arm Neoverse N1 and Apple M1
- Application to: complex FFT, ML-KEM NTT, ML-DSA NTT, X25519 scalar multiplication
  $\Longrightarrow$ Performance on par or faster than state of the art

Contributions

- SLOTHY: Register allocation, instruction scheduling, and software pipelining as a constraint satisfaction problem
- Implementation of SLOTHY using CP-SAT from Google's OR-Tools
- Architecture models: MVE and Neon (partial)
- μArchitecture models: Cortex-M55, Cortex-M85, Cortex-A55, Cortex-A72
  - By now: Arm Neoverse N1 and Apple M1
- Application to: complex FFT, ML-KEM NTT, ML-DSA NTT, X25519 scalar multiplication
  $\Longrightarrow$ Performance on par or faster than state of the art

Contributions

- SLOTHY: Register allocation, instruction scheduling, and software pipelining as a constraint satisfaction problem
- Implementation of SLOTHY using CP-SAT from Google's OR-Tools
- Architecture models: MVE and Neon (partial)
- μArchitecture models: Cortex-M55, Cortex-M85, Cortex-A55, Cortex-A72
  - By now: Arm Neoverse N1 and Apple M1
- Application to: complex FFT, ML-KEM NTT, ML-DSA NTT, X25519 scalar multiplication
  $\Longrightarrow$ Performance on par or faster than state of the art

SLOTHY advantages

- Similar or better results
- Support multiple μarchs

- Less time-intensive development
- Algorithm-level experiments easier

- Input to SLOTHY is executable $\Longrightarrow$ eases testing + do not need to learn DSL
- Software pipelining (see later) allows much more compact code
- Eases maintenance, audit, and formal verification

SLOTHY advantages

- Similar or better results
- Support multiple μarchs

- Less time-intensive development
- Algorithm-level experiments easier

- Input to SLOTHY is executable $\implies$ eases testing + do not need to learn DSL
- Software pipelining (see later) allows much more compact code
- Eases maintenance, audit, and formal verification

SLOTHY advantages

- Similar or better results
- Support multiple μarchs

- Less time-intensive development
- Algorithm-level experiments easier

- Input to SLOTHY is executable $\Longrightarrow$ eases testing + do not need to learn DSL
- Software pipelining (see later) allows much more compact code
- Eases maintenance, audit, and formal verification

In-order CPUs vs. out-of-order CPUs

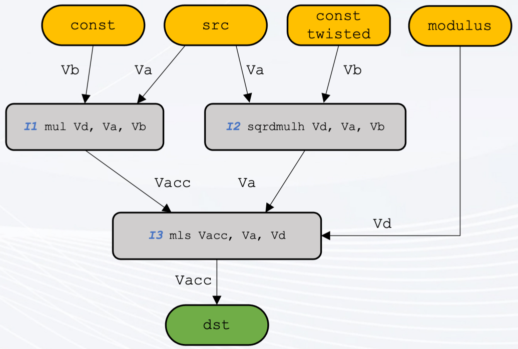
- On **in-order cores** (e.g., A55 or M55) scheduling is **vital for best performance**
- Good scheduling essential **on some OOO** cores too (e.g., Cortex-A72)

Scheduling as constraint solving problem

Input Assembly

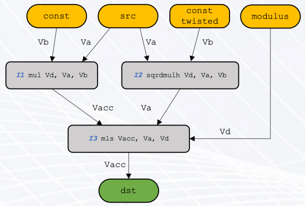
```
// in:  src, modulus,
//      const, const_twisted
mul       dst, src, const
sqrdmulh tmp, src, const_twisted
mls       dst, tmp, modulus
// out: dst
```

Computational Flow Graph



Scheduling as constraint solving problem: Correctness

Computational Flow Graph

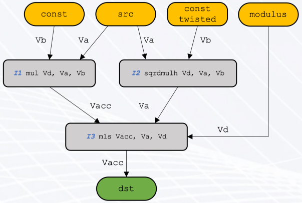


Correctness

- Program position: Integer variables
  for each instruction
  `I1.pos, I2.pos, I3.pos`
- All program positions mutually distinct
- Consumer after producer constraint
  for each edge in the CFG
  `I3.pos > I1.pos`
  `I3.pos > I2.pos`

Scheduling as constraint solving problem: Correctness

Computational Flow Graph

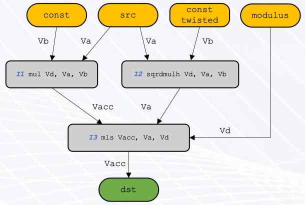


Correctness

- Program position: Integer variables
  for each instruction
  `I1.pos`, `I2.pos`, `I3.pos`
- All program positions mutually distinct
- Consumer after producer constraint
  for each edge in the CFG
  `I3.pos > I1.pos`
  `I3.pos > I2.pos`

Scheduling as constraint solving problem: Correctness

Computational Flow Graph



Correctness

- Program position: Integer variables *for each instruction* `I1.pos, I2.pos, I3.pos`
- All program positions mutually distinct
- Consumer after producer constraint *for each edge in the CFG*

```
I3.pos > I1.pos
I3.pos > I2.pos
```

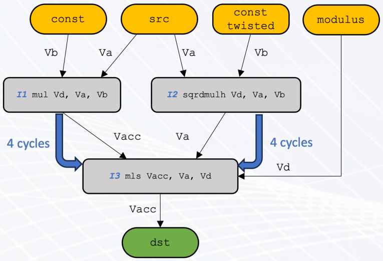
Scheduling as constraint solving problem: Register allocation

- **Boolean variables** (Register is output of instruction)
  `I1.V0`, ..., `I1.V31` `I2.V0`, ..., `I2.V31`
  `I3` need to use same output register as `I1` (input/output)
- **Register allocation constraint**
  Exactly one of `I.V0`, ..., `I.V31` is true
- **Register usage interval** (conditioned on boolean variable)
  `[I1.pos, I3.pos]` `[I2.pos, I3.pos]`
- **Lifetime constraint**
  For each register: Active usage intervals cannot overlap

Scheduling as constraint solving problem: Register allocation

- **Boolean variables** (Register is output of instruction)
  `I1.V0`, ..., `I1.V31` `I2.V0`, ..., `I2.V31`
  `I3` need to use same output register as `I1` (input/output)
- **Register allocation constraint**
  Exactly one of `I.V0`, ..., `I.V31` is true
- **Register usage interval** (conditioned on boolean variable)
  `[I1.pos, I3.pos]` `[I2.pos, I3.pos]`
- **Lifetime constraint**
  For each register: Active usage intervals cannot overlap

Scheduling as constraint solving problem: Latencies



Instruction group	AArch64 instructions	Exec latency	Execution throughput
ASIMD multiply	MUL, SQDMULH, SQRDMULH	4	2*

- μarch constraints

```
I3.pos > I1.pos + 3
I3.pos > I2.pos + 3
```

Scheduling as constraint solving problem: Other aspects

SLOTHY actually models a lot more details

- Allow for **gaps in scheduling** to account for **presence of stalls**
- Model **multi-issue CPUs** and issuing constraints (essential for e.g., dual-issue Cortex-A55)
- Model **execution units** and **instruction throughput** (essential for e.g. Cortex-M55 and Cortex-M85)
- Forwarding paths
- Support dependencies through memory (e.g. stack spills)

Scheduling as constraint solving problem: Other aspects

SLOTHY actually models a lot more details

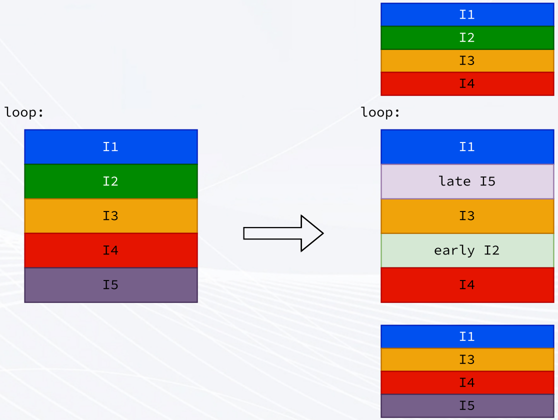
- Allow for **gaps in scheduling** to account for **presence of stalls**
- Model **multi-issue CPUs** and issuing constraints (essential for e.g., dual-issue Cortex-A55)
- Model **execution units** and **instruction throughput** (essential for e.g. Cortex-M55 and Cortex-M85)
- Forwarding paths
- Support dependencies through memory (e.g. stack spills)

Scheduling as constraint solving problem: Other aspects

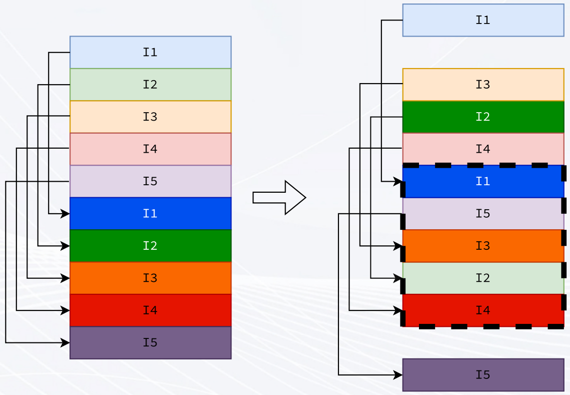
SLOTHY actually models a lot more details

- Allow for **gaps in scheduling** to account for **presence of stalls**
- Model **multi-issue CPUs** and issuing constraints (essential for e.g., dual-issue Cortex-A55)
- Model **execution units** and **instruction throughput** (essential for e.g. Cortex-M55 and Cortex-M85)
- Forwarding paths
- Support dependencies through memory (e.g. stack spills)

Software Pipelining a.k.a. periodic loop-interleaving



Software Pipelining a.k.a. periodic loop-interleaving



SLOTHY: Self-check

SLOTHY performs a simple automatic self-check after optimization

- Transform both input and output code into a data-flow graph
- Check that DFGs are isomorphic
- This is easy given the re-ordering permutation

$\Longrightarrow$ This is not formal verification, but it is very useful!

SLOTHY: Self-check

SLOTHY performs a simple automatic self-check after optimization

- Transform both input and output code into a data-flow graph
- Check that DFGs are isomorphic
- This is easy given the re-ordering permutation

$\Longrightarrow$ This is not formal verification, but it is very useful!

Selected results: NTTs on Cortex-M55 and Cortex-M85

			Cortex-M55		Cortex-M85	
		Type	Cycles	Code size	Cycles	Code size
32-bit ML-DSA NTT	[BBMK+21]	Scripted ASM	2017	7.8 KB	1980	7.8 KB
	Our work	Clean	3602	1.0 KB	3350	1.0 KB
	3+3+2 layers	**`slothy`**	2037	1.1 KB	1997	1.1 KB
16-bit ML-KEM NTT	Our work	Clean	1619	0.7 KB	1511	0.7 KB
	2+3+2 layers	**`slothy`**	942	1.0 KB	910	1.0 KB

- Comparing to **`https://eprint.iacr.org/2021/998`**
- Comparable speed to handwritten MVE assembly (only available for ML-DSA)
- $7\times$ smaller code size due to software pipelining

Selected results: NTTs on Cortex-A55 and Cortex-A72

			Cortex-A55		Cortex-A72	
		Type	Cycles	Code size	Cycles	Code size
32-bit ML-DSA NTT	[BHK+21]	Handwritten ASM	2436	2.3 KB	2241	2.3 KB
	Our work	Clean	3542	1.5 KB	2250	1.1 KB
	3+5 layers	**`slothy`**	1728	2.8 KB	1766	2.1 KB
16-bit ML-KEM NTT	[BHK+21]	Handwritten ASM	1245	2.7 KB	1200	2.7 KB
	Our work	Clean	1914	1.0 KB	1307	0.8 KB
	3+5 layers	**`slothy`**	891	1.9 KB	932	1.4 KB

- Comparing to **`https://eprint.iacr.org/2021/986`**
- More compact code than state-of-the-art
- Faster code on out-of-order Cortex-A72
- Faster code on in-order Cortex-A55

Selected results: X25519

			Cortex-A55		Cortex-A53	
		Type	Cycles	Code-size	Cycles	Code-Size
X25519	[Len19]	Handwritten ASM	143 849	5.8 KB	144 168	5.8 KB
	Our work	Clean	265 739	5.8 KB	270 186	5.8 KB
		slothy	139 752	5.8 KB	140 096	5.8 KB

- Comparing to hybrid scalar/vector implementation from `https://github.com/Emill/X25519-AArch64`
- Faster on original target platform (Cortex-A53/Raspberry Pi3)
- Faster on Cortex-A55
- Much cleaner code $\Longrightarrow$ No manual interleaving of scalar and vector code needed

Conclusion

- SLOTHY can help you to write faster assembly with less work
  - Try it today: `https://github.com/slothy-optimizer/slothy`
- We support various μarchs already
  - Arm A-profile: Cortex-A55, Cortex-A72, Neoverse N1, Apple M1
  - Arm M-profile: Cortex-M55, Cortex-M85
- Code studied in this work
  - ML-KEM + ML-DSA NTTs
  - Complex FFTs
  - X25519
- We use SLOTHY for our own projects
  - More crypto coming soon
  - We will keep adding needed features

Conclusion

- SLOTHY can help you to write faster assembly with less work
    - Try it today: `https://github.com/slothy-optimizer/slothy`
- We support various μarchs already
    - Arm A-profile: Cortex-A55, Cortex-A72, Neoverse N1, Apple M1
    - Arm M-profile: Cortex-M55, Cortex-M85
- Code studied in this work
    - ML-KEM + ML-DSA NTTs
    - Complex FFTs
    - X25519
- We use SLOTHY for our own projects
    - More crypto coming soon
    - We will keep adding needed features

Conclusion

- SLOTHY can help you to write faster assembly with less work
  - Try it today: **`https://github.com/slothy-optimizer/slothy`**
- We support various μarchs already
  - Arm A-profile: Cortex-A55, Cortex-A72, Neoverse N1, Apple M1
  - Arm M-profile: Cortex-M55, Cortex-M85
- Code studied in this work
  - ML-KEM + ML-DSA NTTs
  - Complex FFTs
  - X25519
- We use SLOTHY for our own projects
  - More crypto coming soon
  - We will keep adding needed features

Conclusion

- SLOTHY can help you to write faster assembly with less work
    - Try it today: `https://github.com/slothy-optimizer/slothy`
- We support various μarchs already
    - Arm A-profile: Cortex-A55, Cortex-A72, Neoverse N1, Apple M1
    - Arm M-profile: Cortex-M55, Cortex-M85
- Code studied in this work
    - ML-KEM + ML-DSA NTTs
    - Complex FFTs
    - X25519
- We use SLOTHY for our own projects
    - More crypto coming soon
    - We will keep adding needed features

New results in AWS' s2n-bignum (not done by us)

- Our X25519 got formally verified (HOL Light)
  - Integrated into s2n-bignum (used by AWS-LC)
  - Graviton 2: Scalar multiplication 74% faster than previous code in AWS-LC (X25519 keygen+scalar multiplication $> 3\times$ faster than OpenSSL)
  - Formal verification is made easier by using SLOTHY
    1. Prove SLOTHY input correct
    2. Prove optimized code is still correct
- More formally-verified ECC
  - P256, P384, P521 $\Longrightarrow$ Integrated into s2n-bignum (used by AWS-LC)
  - FV in two steps: Prove input correct + **automated equivalence check in HOL Light**
- RSA
  - Blogpost: **Formal verification makes RSA faster — and faster to deploy**
    `https://www.amazon.science/blog/formal-verification-makes-rsa-faster-and-faster-to-deploy`

New results in AWS' s2n-bignum (not done by us)

- Our X25519 got formally verified (HOL Light)
  - Integrated into s2n-bignum (used by AWS-LC)
  - Graviton 2: Scalar multiplication 74% faster than previous code in AWS-LC (X25519 keygen+scalar multiplication $> 3\times$ faster than OpenSSL)
  - Formal verification is made easier by using SLOTHY
    1. Prove SLOTHY input correct
    2. Prove optimized code is still correct
- More formally-verified ECC
  - P256, P384, P521 $\Longrightarrow$ Integrated into s2n-bignum (used by AWS-LC)
  - FV in two steps: Prove input correct + **automated equivalence check in HOL Light**
- RSA
  - Blogpost: **Formal verification makes RSA faster — and faster to deploy**
    `https://www.amazon.science/blog/formal-verification-makes-rsa-faster-and-faster-to-deploy`

New results in AWS' s2n-bignum (not done by us)

- Our X25519 got formally verified (HOL Light)
  - Integrated into s2n-bignum (used by AWS-LC)
  - Graviton 2: Scalar multiplication 74% faster than previous code in AWS-LC (X25519 keygen+scalar multiplication $> 3\times$ faster than OpenSSL)
  - Formal verification is made easier by using SLOTHY
    1. Prove SLOTHY input correct
    2. Prove optimized code is still correct
- More formally-verified ECC
  - P256, P384, P521 $\Longrightarrow$ Integrated into s2n-bignum (used by AWS-LC)
  - FV in two steps: Prove input correct + **automated equivalence check in HOL Light**
- RSA
  - Blogpost: Formal verification makes RSA faster — and faster to deploy `https://www.amazon.science/blog/formal-verification-makes-rsa-faster-and-faster-to-deploy`

New results in AWS' s2n-bignum (not done by us)

- Our X25519 got formally verified (HOL Light)
  - Integrated into s2n-bignum (used by AWS-LC)
  - Graviton 2: Scalar multiplication 74% faster than previous code in AWS-LC (X25519 keygen+scalar multiplication $> 3\times$ faster than OpenSSL)
  - Formal verification is made easier by using SLOTHY
    1. Prove SLOTHY input correct
    2. Prove optimized code is still correct
- More formally-verified ECC
  - P256, P384, P521 $\Longrightarrow$ Integrated into s2n-bignum (used by AWS-LC)
  - FV in two steps: Prove input correct + **automated equivalence check in HOL Light**
- RSA
  - Blogpost: **Formal verification makes RSA faster — and faster to deploy**
    `https://www.amazon.science/blog/formal-verification-makes-rsa-faster-and-faster-to-deploy`

Ongoing work

- Keccak for AArch64
  - Our previous work: **Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64**, `https://eprint.iacr.org/2022/1243`
  - SLOTHY can automate the majority of this work
  - Very useful for FIPS203, FIPS204, and FIPS205, too
- Arm Cortex-M7
  - Dual-issue CPU implementing Armv7E-M
  - Existing Cortex-M4 code performs very poorly
  - We are looking at Keccak and ML-KEM + ML-DSA NTTs
- Register spilling
  - Symbolic registers $\Longrightarrow$ SLOTHY will find a register allocation
  - Before: Fixed instruction $\Longrightarrow$ If there are not enough registers, SLOTHY will fail
  - Recently added support for rudimentary register spilling

Ongoing work

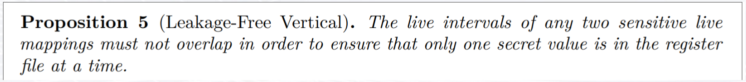
- Keccak for AArch64
  - Our previous work: **Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64**, `https://eprint.iacr.org/2022/1243`
  - SLOTHY can automate the majority of this work
  - Very useful for FIPS203, FIPS204, and FIPS205, too
- Arm Cortex-M7
  - Dual-issue CPU implementing Armv7E-M
  - Existing Cortex-M4 code performs very poorly
  - We are looking at Keccak and ML-KEM + ML-DSA NTTs
- Register spilling
  - Symbolic registers $\Longrightarrow$ SLOTHY will find a register allocation
  - Before: Fixed instruction $\Longrightarrow$ If there are not enough registers, SLOTHY will fail
  - Recently added support for rudimentary register spilling

Ongoing work

- Keccak for AArch64
  - Our previous work: **Hybrid scalar/vector implementations of Keccak and SPHINCS+ on AArch64**, `https://eprint.iacr.org/2022/1243`
  - SLOTHY can automate the majority of this work
  - Very useful for FIPS203, FIPS204, and FIPS205, too
- Arm Cortex-M7
  - Dual-issue CPU implementing Armv7E-M
  - Existing Cortex-M4 code performs very poorly
  - We are looking at Keccak and ML-KEM + ML-DSA NTTs
- Register spilling
  - Symbolic registers $\Longrightarrow$ SLOTHY will find a register allocation
  - Before: Fixed instruction $\Longrightarrow$ If there are not enough registers, SLOTHY will fail
  - Recently added support for rudimentary register spilling

Future work

SLOTHY could automate the application of constraints avoiding pipeline leakage.
$\Longrightarrow$ Promising future direction for research


Proposition 5 (Leakage-Free Vertical)**.** *The live intervals of any two sensitive live mappings must not overlap in order to ensure that only one secret value is in the register file at a time.*

(PoMMES: Prevention of Micro-architectural Leakages in Masked Embedded Software – CHES 2024)

Thank you very much for your attention!

matthias@chelpis.com

`https://github.com/slothy-optimizer/slothy`

`https://eprint.iacr.org/2022/1303`

Results: Complex FFTs

		Type	Cortex-M55 $\frac{\text{Cycles}}{\text{Butterfly}}$	Cortex-M85 $\frac{\text{Cycles}}{\text{Butterfly}}$
CFFT	Q.31	Intrinsics	30 (+16%)	29 (+13%)
		Handwritten	28 (+10%)	26 (+3%)
		`slothy`	25	25
CFFT	FP32	Intrinsics	33 (+15%)	34 (+20%)
		Handwritten	29 (+3%)	29 (+6%)
		`slothy`	28	27

- MVE intrinics and assembly of fixed- and floating-point FFTs from Arm EndpointAI
- SLOTHY finds stall-free scheduling $\Longrightarrow$ 3 - 10% fewer cycles
- M55: 7 early instructions; M85: 1 early instruction
- **Sent the optimized code to Arm $\Longrightarrow$ Now merged into EndpointAI**