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

Amin Abdulrahman[1,2], Hanno Becker[3],
Matthias J. Kannwischer[4] and Fabien Klein[5]

[1] Ruhr University Bochum, Germany
amin.abdulrahman@mpi-sp.org
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
[3] Automated Reasoning Group, Amazon Web Services, Cambridge, United Kingdom[†]
beckphan@amazon.co.uk
[4] Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu
[5] Arm Limited
fabien.klein@arm.com

Abstract. Handwritten assembly is a widely used tool in the development of high-performance cryptography: By providing full control over instruction selection, instruction scheduling, and register allocation, highest performance can be unlocked. On the flip side, developing handwritten assembly is not only time-consuming, but the artifacts produced also tend to be difficult to review and maintain – threatening their suitability for use in practice.

In this work, we present SLOTHY (Super (Lazy) Optimization of Tricky Handwritten assembly), a framework for the automated super optimization of assembly with respect to instruction scheduling, register allocation, and loop optimization (software pipelining): With SLOTHY, the developer controls and focuses on algorithm and instruction selection, providing a readable "base" implementation in assembly, while SLOTHY automatically finds optimal and traceable instruction scheduling and register allocation strategies with respect to a model of the target (micro)architecture.

We demonstrate the flexibility of SLOTHY by instantiating it with models of the Cortex-M55, Cortex-M85, Cortex-A55 and Cortex-A72 microarchitectures, implementing the Armv8.1-M+Helium and AArch64+Neon architectures. We use the resulting tools to optimize three workloads: First, for Cortex-M55 and Cortex-M85, a radix-4 complex Fast Fourier Transform (FFT) in fixed-point and floating-point arithmetic, fundamental in Digital Signal Processing. Second, on Cortex-M55, Cortex-M85, Cortex-A55 and Cortex-A72, the instances of the Number Theoretic Transform (NTT) underlying CRYSTALS-Kyber and CRYSTALS-Dilithium, two recently announced winners of the NIST Post-Quantum Cryptography standardization project. Third, for Cortex-A55, the scalar multiplication for the elliptic curve key exchange X25519. The SLOTHY-optimized code matches or beats the performance of prior art in all cases, while maintaining compactness and readability.

Keywords: Superoptimization · Constraint Solving · Cryptography · Post-Quantum Cryptography · Armv8.1-M · AArch64 · Helium · Neon · Kyber · Dilithium · X25519 · Fast Fourier Transform · FFT · Number Theoretic Transform · NTT · Software Pipelining · Google OR-Tools

[†]This work was done while the author was at Arm Research.

1 Introduction

Software optimization involves trade-offs, some affecting traditional metrics such as performance, code-size, and memory usage, others affecting softer metrics such as readability, maintainability, and verifiability. While the initial study of algorithms and implementations for nascent computational problems typically focuses on the traditional metrics, it is the softer metrics that are equally important as implementations leave their home of research and step into the "real world". Post-Quantum Cryptography (PQC) is a timely example: Implementations of PQC have been studied extensively in the past years for their core performance metrics. However, now that the winners of the NIST Post-Quantum Cryptography standardization project [NIS16] have been announced and the use of PQC becomes increasingly urgent, the soft metrics become central: Are implementations understandable, maintainable, verifiable? This work aims to facilitate the transition of high-performance PQC implementations from research into practice.

At the architectural and microarchitectural level, implementors often face the choice between handwritten assembly and compiled languages. This choice is a trade-off: While handwritten assembly typically yields highest performance, it also tends to be time-consuming to develop and difficult to read and maintain. Conversely, writing software in compiled languages, such as C, offers fast development times as well as clean and portable code, but performance is generally worse than that of handwritten assembly. The reason is simple: The use of compiled languages leaves a number of tasks to the compiler which are too complex for compilers to solve optimally, especially within their limited time budget. Solving those complex tasks manually is thus an opportunity for performance, but also a risk for laborious development with inscrutable results.

Among the tasks taken care of by compilers are the following: First, identifying how the semantics of terms in the source language can be expressed through the instructions of the target machine language. For example, a C developer may simply write `a%b` for the reduction of `a` modulo `b`, but finding efficient algorithms to compute such modular reductions and expressing them in terms of the target architecture, are non-trivial questions left to the compiler (developers) this way. Second, once this *instruction selection* step is done, the tasks of *register allocation* and *instruction scheduling*. Continuing the example of modular reduction: An implementation of modular reduction via (say) Barrett reduction might have been provided for the target architecture, but if numerous such reductions need to be orchestrated alongside other code, finding appropriate instructions scheduling and register allocation strategies is a difficult problem on its own.

Two popular approaches addressing the dilemma of choosing between handwritten assembly and compiled languages are the following: First, inline assembly allows developers to force the use of a particular instruction sequence while leaving the allocation of input/output registers to the compiler. Second, extensions like the Arm C Language Extension (ACLE) use *intrinsincs* to expose architectural details such as SIMD instructions into higher level languages. Both approaches give the programmer some control over instruction selection, but leave scheduling and/or register allocation to the compiler.

Another approach is to replace or augment off-the-shelf compilation with custom tooling handling some of instruction selection, instruction scheduling or register allocation, or related tasks. For example, the `qhasm` tool [Ber] provides a customizable syntax frontend for assembly language which leaves control over instruction selection and scheduling in the hands of the developer, but takes care of register allocation. This allows the developer to effectively still write assembly, but using symbolic registers and a custom syntax, thereby achieving artifacts that can be more readable and maintainable than raw assembly. However, instruction scheduling remaining in the hands of the programmer is, again, both an opportunity for performance and an impediment to readability and fast development.

In this work, we propose and implement a framework for assembly development in which the developer stays in full control over instruction selection, while instruction

scheduling, register allocation and also loop optimization (aka software pipelining) are automated based on models of the underlying architecture and microarchitecture. This way, manual work is restricted to the most difficult and creative aspect of selecting algorithms and mapping them to the target architecture. Moreover, since register allocation and scheduling are automated, the developer can write "clean" code emphasizing the logic of the computation and using symbolic register names, thereby achieving reviewable and maintainable code artifacts. Finally, since the microarchitectural model is flexible, our approach allows porting code to new microarchitectures with little effort.

Compared with traditional compilation, our approach is less reliant on fast but approximative algorithms, and is instead based on superoptimization in search for *optimal* solutions: More precisely, we use constraint solving to express the simultaneous optimization of register allocation, instruction scheduling, and software pipelining as a constraint satisfaction problem which is then passed to an external solver.

Contributions. Our contributions are fourfold:

1. We describe SLOTHY (Super (Lazy) Optimization of Tricky Handwritten assembly), a (micro)architecture agnostic approach to modelling the simultaneous assembly-level optimization of register allocation, instruction ordering and software pipelining as a constraint satisfaction problem.

2. We develop instantiations of SLOTHY for multiple architectures and microarchitectures: (a) A model of the Armv8.1-M+Helium architecture and the Cortex-M55 + Cortex-M85 microarchitectures implementing it, (b) a model of the AArch64+Neon architecture and the Cortex-A55 + Cortex-A72 microarchitectures implementing it.

3. We provide an implementation `slothy` of SLOTHY based on CP-SAT from Google OR-Tools. As a development tool, `slothy` converts "clean" and microarchitecture-agnostic input assembly (using e.g. symbolic registers and macros for readability) into microarchitecture-specific high-performance assembly. `slothy`'s output is not only fast, but also compact and auditable: Loops can be optimized without unrolling, and we outline how trace annotations can support manual or automatic checks of the equivalence of input and output, without having to trust SLOTHY itself.

4. We demonstrate the practicality of SLOTHY by optimizing multiple real-world workloads from Digital Signal Processing and cryptography: (a) on Cortex-M55 and Cortex-M85, a radix-4 complex Fast Fourier Transform (FFT) in fixed-point and floating-point arithmetic, (b) on Cortex-M55, Cortex-M85, Cortex-A55 and Cortex-A72, instances of the Number Theoretic Transform (NTT) underlying the post-quantum cryptography key encapsulation and signature schemes CRYSTALS-Kyber and CRYSTALS-Dilithium, and (c) on Cortex-A55, the X25519 scalar multiplication. In all cases, `slothy` transforms "clean" base implementations into high-performance implementations that match or outperform prior microarchitecture-specific handwritten assembly, while also maintaining compactness, auditability, and portability to further microarchitectures.

Future work. There are multiple avenues of future work.

First, we encourage research in the application of SLOTHY to other (micro)architectures and workloads. In the context of post-quantum cryptography, for example, we do deliberately not build complete implementations of CRYSTALS-Kyber and CRYSTALS-Dilithium. We believe that doing so, under consideration of the numerous implementation techniques available, would make for an attractive piece of follow-up research. See also Section 4.4.

Second, the soundness & trust story for SLOTHY should be further elaborated: We argue in Section 3.16 that neither SLOTHY nor `slothy` need to be trusted to verify

the optimizations they yield, and outline and partially implement a verification strategy. However, developing that strategy into a full automated verification of the results of SLOTHY would require further research.

Third, integration of SLOTHY with other compilation and/or superoptimization tooling such as [SCC+17] could be considered.

Structure. In Section 2, we discuss various preliminaries on (micro)architecture, compilation and superoptimization. In Section 3, which is the heart of this paper, we describe our approach SLOTHY to the simultaneous optimization of register allocation, instruction scheduling and software pipelining via constraint solving. In Section 4, we discuss our instantiations of SLOTHY for Cortex-M55, Cortex-M85 (both Armv8.1-M+Helium) and Cortex-A55 and Cortex-A72 (both AArch64+Neon). Section 5 discusses our CP-SAT-based implementation of SLOTHY in Python, and illustrates in a toy example how we see it being used as a development tool. Finally, in Sections 6.1, 6.2, 6.3 and 6.4, we work through the examples of the Fast Fourier Transform, Number Theoretic Transform and X25519. We conclude with Section 7 containing some reflections and outlook.

Software. SLOTHY and its instantiations for the aforementioned microarchitectures are available under MIT license on `https://github.com/slothy-optimizer/slothy`.

Related work. There is a rich literature on superoptimization in general and the potential to use Integer Linear Programming (ILP) for it in particular [NG07, Mas87, KL99, WGB94, GW96, KW98, SSA13]. Yet, to the best of our knowledge, our approach to using constraint solving for *simultaneously* addressing assembly-level instruction scheduling, register allocation, and software pipelining, is new. [NG07] handles simultaneous register allocation and the generation and scheduling of spill code within software pipelined loops, but already assumes a scheduling for the loop itself. Our work is complementary and potentially combinable with approaches to superoptimization at other compilation stages, such as [SCC+17] performing peephole superoptimization at the level of the LLVM-IR.

2 Preliminaries

2.1 Compilation terms & techniques

Computational flow graphs. A *computational flow graph* (CFG) represents a computation as a labelled directed graph, with nodes corresponding to computation steps and labelled edges indicating how outputs of one step are used as inputs of others. In this paper, we exclusively work with CFGs representing computations in *assembly*, with nodes corresponding to the instructions of the architecture under consideration. Inputs and outputs to the entire computation are represented as 'virtual' instruction nodes without inputs and outputs, respectively. Figure 1 provides an example. Note that the ordering of instructions and the use of registers are forgotten when *lifting* assembly into a CFG. The reverse process is called *lowering* and is essential in the code generation phase of compilers.

Phase ordering problem. Lowering a CFG into assembly requires *instruction scheduling* and *register allocation*: Instruction scheduling assigns a linear order to instructions — a labelling of the CFG's *nodes*. Register allocation assigns architectural registers to instruction inputs/outputs — a labelling of the CFG's *edges*. Instruction scheduling and register allocation influence each other, as the choice of scheduling restricts the set of valid register allocations, and vice versa: An instruction must not overwrite a register (an aspect of register allocation) if that instruction is placed in between a producer and consumer

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

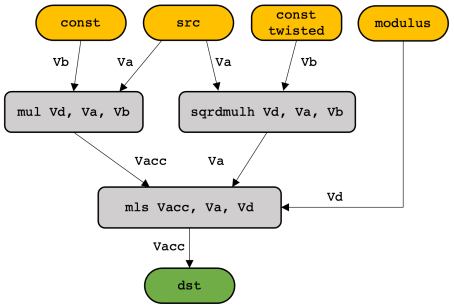

Figure 1: Left: Barrett multiplication [BHK+21] in Neon (lane widths elided). Right: The underlying computational flow graph.

```
vldrw.u32  q0, [inA]
vadd.u32   q0, q0, q0
vadd.u32   q0, q0, q0
vstrw.u32  q0, [inA]
vldrw.u32  q0, [inB]
vmul.u32   q0, q0, q0
vmul.u32   q0, q0, q0
vstrw.u32  q0, [inB]
```

```
vldrw.u32  q0, [inA]
vadd.u32   q0, q0, q0
vadd.u32   q0, q0, q0
vstrw.u32  q0, [inA]
vldrw.u32  q1, [inB]
vmul.u32   q1, q1, q1
vmul.u32   q1, q1, q1
vstrw.u32  q1, [inB]
```

```
vldrw.u32  q0, [inA]
vadd.u32   q0, q0, q0
vldrw.u32  q1, [inB]
vadd.u32   q0, q0, q0
vmul.u32   q1, q1, q1
vstrw.u32  q0, [inA]
vmul.u32   q1, q1, q1
vstrw.u32  q1, [inB]
```

Listing 1: Left: Two logically independent code paths using the same register. Middle: Register renaming to separate register usage and enable interleaving. Right: Interleaving.

of said register (an aspect of scheduling). The relation and ordering between instruction scheduling and register allocation is called the *phase ordering problem*.

Software pipelining. Software pipelining [Lam88, RG81] is a software optimization technique whereby multiple iterations of a loop are interleaved to create instruction level parallelism and thereby facilitate execution on the underlying microarchitecture. Popular approaches are iterative modulo scheduling [Rau94] and swing modulo scheduling [LGAV96]. While originally devised for Very Long Instruction Word (VLIW) processors, software pipelining is well-known also for classical microarchitectures: When the execution of one loop iteration cannot progress due to latency constraints or lack of availability of functional units, instructions from the next iteration(s) may be pulled forward to fill the gaps. This is conceptually similar to how out-of-order microarchitectures reorder instructions during execution (explained below), but explicit software pipelining may still be beneficial even for such microarchitectures.

Software pipelining puts pressure on the register file since iterations may only be interleaved once there is no collision in their register use. In an out-of-order microarchitecture, this is a consequence of register renaming assigning a fresh physical register to each instruction output, thereby benefiting from a physical register file that's larger than the architectural register file. Software pipelining, however, has to perform manual register renaming within the *architectural* register file, which can be challenging. See Listing 1 for a toy example of an interleaving opportunity created through register renaming.

2.2 (Micro)architecture

While the ideas behind SLOTHY are (micro)architecture independent, a basic knowledge of CPU (micro)architecture in general, and the Arm architecture in particular, is necessary to appreciate the concrete examples discussed in this paper. We provide a brief overview.

2.2.1 Microarchitecture 1-1

This section provides a minimal mental model of instruction execution, formulated in a somewhat non-standard way via computational flow graphs. For thorough introduction to microarchitecture design, we refer to [TA16].

An instruction's life in stages. All processors considered here execute instructions in a series of stages called pipeline. At the start of the pipeline, instructions are fetched from memory, decoded and have their arguments determined. The processor then tracks the availability of an instruction's arguments, and when ready, *issues* it to an *execution unit*: For example, arithmetic operations are typically sent to an arithmetic logical unit (ALU). When the instruction's results are ready, they are written back to the register file or forwarded to instructions waiting for them.

Execution order. While the journey of an individual instruction through the pipeline is sequential, the above does not imply an ordering between *different* instructions with respect to their time of issuing and execution. Instead, in the language of computational flow graphs (CFG), we can (conceptually) think of the processor as conducting an ad-hoc lifting of the instruction stream into a CFG and then dynamically lowering it. The distinction of *in-order* vs. *out-of-order* execution is about how this lowering is determined.

In-order execution. Recall that lowering a CFG requires assigning output registers and an order to instructions. In-order execution is the simplest possible approach here: The issue order is the program order, and the physical registers used by an instruction directly correspond to the architectural registers used in the assembly.

Out-of-order execution. Out-of-order execution lowers the instruction stream's CFG by availability: An instruction is issued when arguments and execution unit(s) are available. If multiple instructions are ready in this sense, typically the earliest instruction with respect to the original program order is issued. Instructions "waiting" in the CFG are typically realized via multiple *issue queues* connecting instructions to the units that will execute them. An instruction entering an issue queue is called its *dispatch stage*, and the part of the pipeline prior to dispatch is called the *frontend*; it is typically in-order, and in our mental model corresponds to the dynamic construction of the CFG.

Register renaming. In out-of-order execution, the physical registers used by instructions are decoupled from the architectural registers used in the original assembly — the latter are merely dependency labels. Instead, an instruction is assigned "fresh" physical registers for their outputs at the *register renaming* stage prior to dispatch. In particular, in an out-of-order CPU, the left and middle snippets from Listing 1 are identical after register renaming, enabling an out-of-order execution as in the right snippet. Without register renaming, out-of-order execution would be severely limited by the resulting register hazards.

2.2.2 M-Profile Vector Extension / Helium

The M-Profile Vector Extension (MVE) is a Single Instruction Multiple Data (SIMD) extension that was introduced as part of the Armv8.1-M architecture [Armf]. Its primary goal is to enable higher performance for signal processing and machine learning applications.

MVE is also referred to as Arm® Helium™ technology, in alignment with the Arm® Neon™ technology architecture extension for A-profile processors [Arma, Section C.3.5], or Neon for short. However, despite the similarity in name, Helium is a new ground-up architecture designed specifically for the tight area/power constraints of the embedded

market. We refer to [BBMK+21, Armg, Arms, Armt] for introductions to Armv8.1-M+Helium and to the reference manual [Armf] for the details.

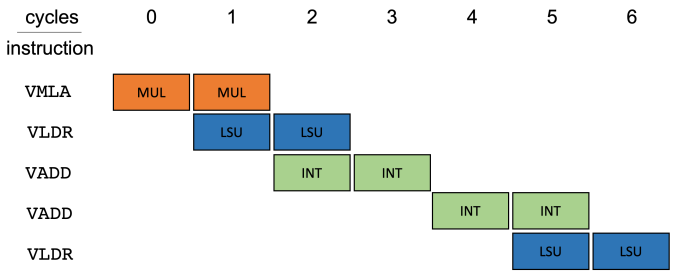

Figure 2: Illustration of instruction overlapping for instructions relying on separate functional units.

What is most important about Helium for the sake of this work is how it carefully introduces software constraints to lower hardware complexity and thereby retain suitability for embedded microcontrollers: Most notably, instruction overlapping and the compact vector register file of 8×128-bit vector registers.

First, instruction overlapping allows to achieve up to $2\times$ performance with the same execution resources compared to non-overlapping, single-issued execution. However, it requires instructions to be scheduled in such a way that they run on different functional units and are therefore amenable to overlapping. Figure 2 provides an illustration, assuming an implementation of Helium where each vector instruction takes two cycles ("dual beat implementation") and where there are separate functional units for vector load/store (LSU), integer addition/logical operations (INT), and multiply operations (MUL) — the Cortex-M55 CPU is an example. We can see how the first instructions overlap, leading to high resource utilization, but how the consecutive pair of `VADD` stalls the pipeline. Typically, there is flexibility in the instruction scheduling, so that good interleaving of different instruction types is possible. For example, Listing 2 shows two versions of the same piece of (meaningless) Helium assembly, one poorly scheduled, making little use of instruction overlapping, and another with a good mix of instructions, facilitating overlapping.

Second, the compact vector register file reduces the cost of CPUs implementing Helium, but requires developers or compilers to manage register usage very carefully, and balance it with the general purpose register file through the use of scalar-vector instructions.

A primary goal of this work is to demonstrate how to automate the process of solving the software constraints posed by Helium through constraint solving.

Implementations. Armv8.1-M+Helium is implemented by the Cortex-M55 and Cortex-M85 CPUs, both of which we will be optimizing for in this paper. The Cortex-M55 is the first processor supporting Armv8.1-M+Helium, and considered a mid-range implementation in Arm's M-profile line. The Cortex-M85 processor, in turn, is considered a high-end implementation, which in addition to support for Helium also offers extensive dual-issuing capabilities. More relevant microarchitectural details will be discussed in Section 4.2.

2.2.3 AArch64+Neon

We will also be optimizing workloads for the Cortex-A55 and Cortex-A72 CPUs. Those CPUs belong to the A-profile of the Arm architecture and implement the AArch64 64-bit execution state, which was introduced in Armv8-A [Armb] and is also used in the Armv8-R and Armv9-A architectures. AArch64 offers 31 general purpose registers of 64-bit each, as well as 32 vector registers of 128-bit each, which are operated on by the A64

```
1  vldrw.u32  q0, [inA]
2  vldrw.u32  q1, [inA, #16]  //-
3  vldrw.u32  q2, [inA, #32]  //-
4  vldrw.u32  q7, [inB], #16  //-
5  vmulh.u32  q0, q0, q7
6  vmulh.u32  q1, q1, q7      //-
7  vmulh.u32  q2, q2, q7      //-
8  vadd.u32   q0, q0, q0
9  vadd.u32   q0, q0, q7      //-
10 vadd.u32   q1, q1, q1      //-
11 vadd.u32   q1, q1, q7      //-
12 vadd.u32   q2, q2, q2      //-
13 vadd.u32   q2, q2, q7      //-
14 vstrw.u32  q1, [inA, #16]
15 vstrw.u32  q2, [inA, #32]  //-
16 vstrw.u32  q0, [inA], #48  //-
```

```
1  vldrw.u32 q1, [inB], #16
2  vldrw.u32 q2, [inA, #16]  //-
3  vmulh.u32 q7, q2, q1
4  vldrw.u32 q4, [inA, #32]
5  vadd.u32  q7, q7, q7
6  vmulh.u32 q6, q4, q1
7  vadd.u32  q4, q7, q1
8  vldrw.u32 q2, [inA]
9  vadd.u32  q7, q6, q6
10 vmulh.u32 q6, q2, q1
11 vadd.u32  q7, q7, q1
12 vstrw.u32 q4, [inA, #16]
13 vadd.u32  q4, q6, q6
14 vstrw.u32 q7, [inA, #32]
15 vadd.u32  q7, q4, q1
16 vstrw.u32 q7, [inA], #48
```

Listing 2: Left: Poorly written snippet of Helium assembly with little potential for instruction overlapping. Right: Improved scheduling + register allocation. An //- annotation indicates a structural hazard preventing instruction overlapping.

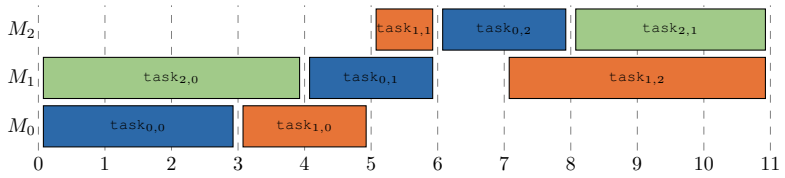

Figure 3: An instance of the job shop problem. $\texttt{task}_{i,j}$ denotes task j of job i.

instruction set including the Neon SIMD vector extension (there are also the SVE and SVE2 vector extensions operating on vectors of flexible width of up to 2048-bits, but we will not consider those in this paper). For an introduction to AArch64+Neon, we refer the reader to [Armp, Armb]. In this paper, we will be optimizing for the Cortex-A55 and Cortex-A72 CPUs: The Cortex-A55 CPU is an efficiency-focused, in-order implementation of AArch64+Neon with extensive dual-issuing capabilities, while the Cortex-A72 CPU is a high performance, out-of-order core — more details will be discussed in Section 4.3.

2.3 Optimization

(Flexible) Job shop problem. The *job shop problem* is a type of scheduling problem: A number of jobs are given which are to be run on a set of machines. Each job consists of a series of tasks to be performed in-order, and for each task there are constraints on the time they take and the machine they must run on. The job shop problem is to find an assignment of the tasks to the machines which optimizes a given measure [ML93], such as the total time taken. For the original job shop problem, each task must run on a fixed machine. For the *flexible* job shop problem, multiple machines may be suitable for each task [Pin16]. A visualization of an instance of the job shop problem is given in Figure 3.

Superoptimization. The term superoptimization was introduced in [Mas87] as finding "the shortest program that computes the same function as the source program by doing an exhaustive search over all possible programs". Here, we use it more broadly for approaches to software optimization that find *optimal* solutions rather than approximations. Superoptimization can be studied from multiple angles, such its position within the software development and compilation flow, the scope of superoptimization, and techniques for implementation — we briefly comment on them and explain where our approach resides.

First, in terms of positioning, the original [Mas87] studies superoptimization at the

level of assembly. In contrast, the more recent [SCC+17] discusses superoptimization at the level of the LLVM intermediate representation (IR), making it more broadly applicable. Our superoptimizer operates at the level of assembly.

Second, in terms of techniques, numerous approaches have been explored, such as brute force enumeration [Mas87], stochastic search [SSA13], and integer linear programming (ILP) [KL99, WGB94, GW96]. Our approach falls into the last category, using Boolean/integer constraint programming to express and solve the problem of finding optimal code.

Third, we comment on the scope of superoptimization. Following [Emb15, Section 11], one can broadly distinguish two separate optimization phases: First, the search for optimal (e.g. short) sequences of assembly expressing a given, typically loop-free, piece of functionality — this requires awareness of instruction semantics. Second, once instructions have been fixed, the search for an optimal scheduling and register allocation strategy — in contrast to the first approach, this only requires knowledge of the architectural signature of instructions, as well as microarchitectural information like the available functional units, latencies and throughputs. Here, we focus on the second approach: Finding optimal solutions for (simultaneous) instruction scheduling and register allocation. Further, we also include software pipelining into the scope of our superoptimizer.

Google OR-Tools. Google OR-Tools [PF] is a software for combinatorial optimization, tailored at solving problems such as vehicle routing, flows, integer and linear programming, and constraint programming. We find that Google OR-Tools's CP-SAT is well suited to our assembly superoptimization problem, in two ways: First, it's fast. Second, CP-SAT's API allows for the specification of a mix of Boolean, integer, and interval variables, and moreover offers convenient constraints such as non-overlapping for intervals, or mutual difference for a set of integer variables. However, we expect our modeling approach to apply to other constraint solvers as well, and encourage further research and comparison.

2.4 The Fourier Transform and its friends

Introduction. The *Fourier Transform* is a transformation for the decomposition of signals into frequency components. It has numerous incarnations — such as Fourier series, the Number Theoretic Transform, and even harmonic analysis on number fields — and a vast range of applications. Simply put, the importance of the Fourier Transform cannot be overstated. Here, we are interested in the discrete Fourier Transform over the complex numbers, which can be viewed as $\mathbb{C}^n \to \mathbb{C}^n, (x_i) \mapsto (\sum_j x_i \zeta_n^{ij})$ where $\zeta_n = \exp(2\pi i/n)$ is the standard primitive n-th root of unity. Algebraically, this is the same as the splitting by evaluation $(\mathrm{ev}_{\zeta_n^i}) : \mathbb{C}[X]/(X^n - 1) \xrightarrow{\cong} \prod_i \mathbb{C}[X]/(X - \zeta_n^i)$.

Fast Fourier Transform. The *Fast Fourier Transform* (FFT) is a method for the efficient computation of the Fourier Transform. While the above description of the Fourier Transform suggests quadratic complexity, the FFT splits the computation into a logarithmic number of *layers* of linear complexity each, giving an overall complexity of $\mathcal{O}(n \log n)$. Each FFT layer operates on strided blocks of r data units via so-called *butterflies*, and r is called the *radix* of the FFT. Common choices are radix-2 and radix-4. Efficient implementations of the FFT are an essential component of any Digital Signal Processing (DSP) library, with complex numbers presented in either floating point or fixed point format.

In the realm of Post-Quantum Cryptography, the FFT is of interest for the fast Fourier sampling of the digital signature scheme Falcon [PFK+22], selected for standardization by NIST in 2022 [NIS16].

Number Theoretic Transform. The *Number Theoretic Transform* (NTT) is a variant of the Fourier Transform that's defined over the integers rather than the complex numbers:

While the latter splits $\mathbb{C}[X]/(X^n - 1)$ as $\prod_i \mathbb{C}[X]/(X - \zeta_n^i)$ for the complex n-th root of unity $\zeta_n = \exp(2\pi i/n)$, the NTT splits a modular polynomial ring $\mathbb{F}_q[X]/(X^n - 1)$ as $\prod_i \mathbb{F}_q[X]/(X - \omega^i)$, for $\omega \in \mathbb{F}_q$ a *modular* primitive n-th root of unity, that is, $\omega^n = 1$ modulo q, but $\omega^i \neq 1$ modulo q for all $i < n$.

Structurally, the NTT is the same as the Fourier Transform, and in particular can be implemented in $\mathcal{O}(n \log n)$ time through a variant of the Fast Fourier Transform. However, the underlying coefficient arithmetic relies on modular integer arithmetic rather than floating point or fixed point arithmetic.

Fast implementations of the Number Theoretic Transform are essential for high performance implementations of post-quantum cryptography, which use the NTT for polynomial multiplication (using the analogue of the convolution theorem in digital signal processing). The recently designated winners Kyber [SAB+22] and Dilithium [LDK+22] of the NIST Post-Quantum Cryptography standardization project [NIS16] both rely on the NTT. In this work, we target the NTTs for these schemes. Kyber uses a 7-layer incomplete 16-bit NTT with $n = 256$ and $q = 3329$, while Dilithium uses an 8-layer complete 32-bit NTT with $n = 256$ and $q = 2^{23} - 2^{13} + 1$.

Basic aspects of NTT implementations are the merging of layers (depending on register pressure), the growth of coefficients and insertion of modular reductions (depending on the underlying prime), and the underlying primitive for modular multiplication.

3 Assembly optimization as a constraint solving problem

This section, which is the heart of our work, describes SLOTHY — Super (Lazy) Optimization of Tricky Handwritten assembly — our approach to modeling assembly superoptimization as a constraint solving problem.

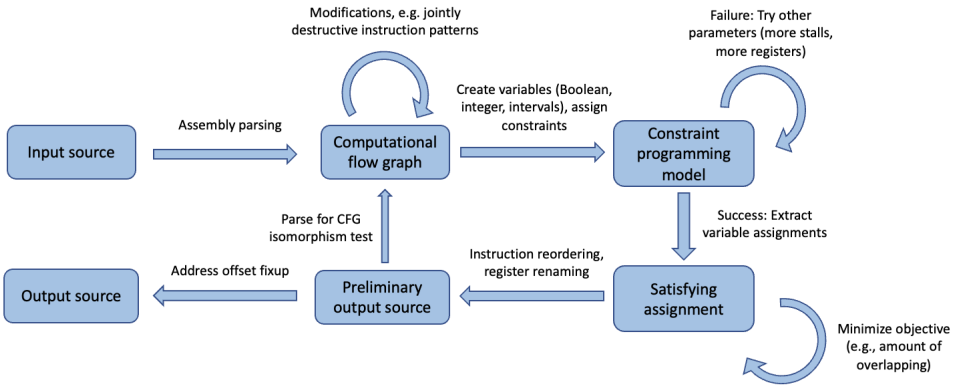

Figure 4: High-level overview of operation of SLOTHY

3.1 Scope

SLOTHY optimizes instruction scheduling, register allocation and software pipelining. That is, it considers reordering of instructions and change of their use of registers in search for some functionally equivalent but optimally performing variant of the code. Importantly, instruction scheduling and register allocation are considered *simultaneously*, avoiding the phase ordering problem (Section 2.1). In case of a loop, SLOTHY also simultaneously searches for suitable interleavings of iterations. Put differently, SLOTHY retains the (isomorphism class of the) source's computational flow graph.

Figure 4 provides an overview of SLOTHY's operation: First, the input is parsed and converted to a computational flow graph. Next, a constraint model is derived as explained below, and passed to an external solver. Upon success, the (optimal) satisfying assignment found by the solver is converted back into the output source. This source is then subject to some post-processing and self-check before being returned to the caller.

SLOTHY's core does not change instructions except for register renaming. In fact, it has no knowledge of the semantics of instructions beyond their signatures, that is, the number and types of inputs, outputs, and input/outputs. It remains the responsibility of the developer to find ways to express the target computation in terms of the underlying architecture. This is a core design choice of SLOTHY: The search for algorithms and instruction sequences is a very different problem from that of instruction ordering and register renaming, and one where we believe human originality brings most to the table. There *are* ways ("peephole optimization" [McK65, SCC+17]) to automate to the search for highly efficient instruction patterns for basic blocks of functionality — however, due to the different nature of the optimization and their high complexity already for small blocks of code, we believe that peephole optimization phases should come prior to the optimization of instruction scheduling and register renaming addressed by SLOTHY.

Finally, SLOTHY has no built-in notion of memory, only registers. First, reasoning about *dynamic* memory accesses is not possible without awareness of the precise semantics of instructions, which is out of scope of SLOTHY by design, as just discussed. Reasoning about *static* memory accesses, such as stack spills, is possible by modelling static memory locations as special "registers" — this will be explained in Section 3.12.

3.2 Overview

The following sections assume that a piece of assembly to be optimized has been provided and parsed into a computational flow graph (see Section 2). Our goal, then, is to define a constraint model — that is, a set of integer, boolean, and interval variables together with a set of conditions imposed upon them — whose set of solutions captures lowerings of the CFG which are both valid (with respect to the underlying architecture) and of high performance (with respect to the underlying microarchitecture).

Aspects of the model that pertain to validity are discussed in Sections 3.3 and 3.4, while Sections 3.6 and 3.8 discuss microarchitectural performance constraints. Section 3.5 discusses how we incorporate software pipelining into the constraint model, and Sections 3.13, 3.12, 3.7 and 3.14 discuss the modelling of memory, stalls, and optimization objectives. Sections 3.15 and 3.16 comment on the performance fidelity and soundness of SLOTHY.

3.3 Correctness/Architecture constraints: Instruction scheduling

Every instruction `I` of the input source is assigned an integer variable `I.pos` defining where the instruction is placed in the output code. To get a unique program order in the output, we require $\{\texttt{I.pos}\}_{\texttt{I}}$ to be mutually distinct. Further, to maintain functional correctness, consumers must come after producers: If instruction `J` consumes output `O` produced by instruction `I` — that is, we have an edge $\texttt{I} \xrightarrow{\texttt{O}} \texttt{J}$ in the computational flow graph — then $\texttt{I.pos} < \texttt{J.pos}$.

3.4 Correctness/Architecture constraints: Register Allocation

For every instruction `I`, every output `O` of `I`, and every possible register `R` that `I` can use for `O`, we assign a Boolean variable `alloc(I,O,R)` indicating whether the instruction `I` uses register `R` for the output `O`. In analogy with the operation of out-of-order microarchitectures, we call this process *register renaming*. In contrast to out-of-order microarchitectures, however, simultaneous input/output arguments are not subject to register renaming as we

need to preserve the architectural constraint that input and output use the same register. For those arguments, we use the notation `alloc(I,O,R)` as a shorthand for `alloc(K,U,R)`, where `K,U` is the transitively computed source of `O`, with `U` being a pure *output* (not merely an input/output). For example, if `I` is an `MLA` in a `{MUL; MLA; ...; MLA}` chain and `O` is the accumulator, we'd always go back to the initial `MUL` which made the choice for which register to use for the accumulator. Finally, note that the input source's choice of registers is irrelevant for register renaming — it is only used initially to construct the computational flow graph from the source.

Multiple constraints need to be satisfied: First, for the uniqueness of register renaming, we require that for fixed `I` and `O`, exactly one Boolean variable in $\{\texttt{alloc(I,O,R)}\}_{\texttt{R}}$ is set. Second, for functional correctness, we need to express that in between a register being produced and consumed, no other instruction uses the register for register renaming. We model this as a disjointness constraint as follows: First, for any instruction `I` and any output or input/output `O`, we add an interval $[\texttt{I} \xrightarrow{\texttt{O}}]$ starting at `I.pos`, and bound its endpoint below by `J.pos` for any $\texttt{I} \xrightarrow{\texttt{O}} \texttt{J}$, as well as by `I.pos` + 1 (in case `O` has no consumers). For any choice of output register `R`, we then add a copy $[\texttt{I} \xrightarrow{\texttt{O}}]_{\texttt{R}}$ of $[\texttt{I} \xrightarrow{\texttt{O}}]$ as a *conditional* interval constrained by `alloc(I,O,R)`. Functional correctness then requires that for any `R`, the set of conditional intervals $\{[\texttt{I} \xrightarrow{\texttt{O}}]_{\texttt{R}}\}_{\texttt{I,O}}$ is non-overlapping.

We note that our modelling of register usage is an instance of the flexible job shop problem, with jobs being instructions and "machines" being registers.

Restricted instructions. Sometimes there are restrictions on the registers that an instruction can use. Such restrictions often apply to individual arguments (such as requiring even, odd or low-half registers), but there are also more complex cases: For example, the output of `VCMUL` in Helium must not coincide with an input, and the four (!) vector arguments to the de-interleaving load `VLD4x` are constrained to the set $\{\texttt{Qi},\texttt{Q(i+1)},\texttt{Q(i+2)},\texttt{Q(i+3)}\}$ for $i = 0, 1, 2, 3, 4$. Individual restrictions on output arguments are straightforwardly added to the register renaming model by restricting the Boolean variables `alloc(I,O,R)` accordingly. To model restrictions for multiple register outputs, such as for `VLD4x`/`VST4x`, we introduce Booleans for the different choices, constrain that precisely one is set, and then add implications to the respective `alloc(I,O,R)` variables.

Jointly destructive instruction patterns. The Armv8.1-M+Helium and AArch64+Neon architectures contain instruction patterns where each instruction individually overwrites only part of the destination register, but where the sequence as a whole overwrites it entirely: In AArch64+Neon, for example, the sequence `{INS Vi[0].2D, Xj; INV Vi[1].2D, Xk}` – writing each of the two 64-bit lanes of a vector in succession — is "jointly destructive" in this sense. In Armv8.1-M+Helium, examples include blocks of `VLD4{0,1,2,3}` as well as pairs of `VQDMLSDH+VQDMLADHX` (jointly performing a complex multiplication in fixed-point arithmetic). By default, the partially destructive components in a jointly destructive instruction pattern would be modelled as having their destination register as an input/output argument, thereby creating an unnecessary data dependency between the instruction pattern and whatever instruction(s) established the previous value of the destination register(s). This, in turn, limits SLOTHY's renaming and reordering flexibility. To address this, SLOTHY allows architecture specifications to modify instructions within the context of an entire computational flow graph, in particular allowing for the retroactive detection and reclassification of input/output registers as a pure output registers in jointly destructive patterns. We leverage this to detect and mark the destination register in aforementioned patterns as a pure output for the first instruction of the sequence, thereby allowing to reorder it past previous instructions which write to the same register.

3.5 Loop interleaving aka Software pipelining

A powerful feature of SLOTHY is software pipelining, that is, interleaving multiple iterations in a loop: Some early instructions, such as initial loads, are moved into the previous iteration, while some late instructions, such as final stores, are deferred to the next iteration. To avoid having to unroll the entire loop, periodicity of code has to be maintained.

Modeling approach. To begin, for each instruction `I` we add three Boolean variables `I.pre`, `I.post` and `I.core`, indicating whether `I` will be an early/late instruction for the previous/next iteration, or whether it stays in its original iteration. Precisely one of {`I.pre`, `I.core`, `I.post`} must be set. For periodicity, we first double the loop body `C`, say as C_1 and C_2, and for an instruction `I`, we denote I_1 and I_2 its copies in C_1 and C_2, respectively. Note that this duplication is internal only and does not enforce an unrolling of the loop in the final code. We then relate I_1 and I_2 through the following periodicity constraints: First, the choices of `I.pre`, `I.core`, `I.post` and of register allocations must be the same for C_1 and C_2: I_1`.{pre,core,post}` $\Leftrightarrow$ I_2`.{pre,core,post}` and `alloc`$(I_1, O, R) \Leftrightarrow$ `alloc`(I_2, O, R) for all `I` $\in$ `C`. Second, "core" instructions are placed exactly n instructions apart, where n is the length of `C`: If `I.core`, then I_2`.pos` $=$ I_1`.pos` $+ n$. Third, and perhaps somewhat non-canonically, we force early instructions in C_1 to become early instructions for the *next* iteration C_3, and late instructions in C_2 to become late instructions for the *previous* iteration C_0. To achieve that, we constrain that for `I.pre` or `I.post`, we have I_2`.pos` $= I_1$`.pos` $- n$ — note the sign! — and "cut" the constraints derived from $I_1 \xrightarrow{O} J_1$ if `I.pre`, and those derived from $I_2 \xrightarrow{O} J_2$ if `J.post`. Despite those "cuts", the required constraints are implied by the remaining periodicity constraints, so long as we impose that for I_1`.pre` and `I` $\xrightarrow{O}$ `J`, we have either `J.pre` or `J.core`, and similar for J_2`.post` — without this condition, dependencies could be entirely and incorrectly removed by never setting `I.core` (a previous version of SLOTHY got this wrong). See Figure 5 for an illustration.

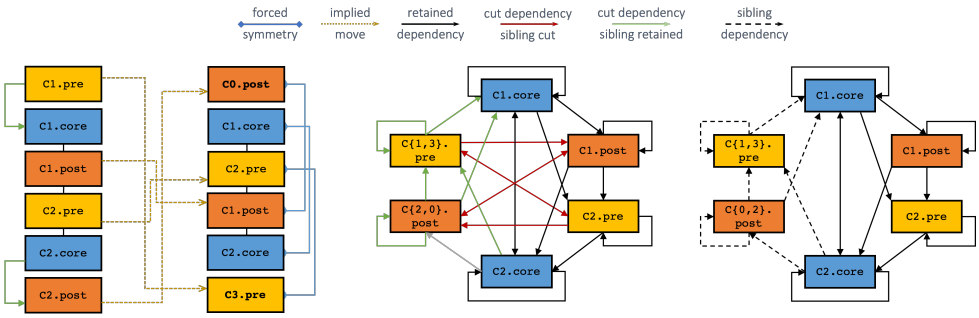

Figure 5: Modelling software pipelining in SLOTHY. Left: Illustration of movement of early/core/late parts forced by periodicity. Middle/Right: Illustration of lost and retained dependencies after applying periodicity.

Loop boundary. We do not model looping instructions such as counter decrements and compare+jumps: First, we don't expect those to have a meaningful impact on the optimization. Second, on Armv8.1-M+Helium in particular, inner iterations in low overhead loops do execute as if the loop had been unrolled, including the potential for instruction overlapping. By modelling the last and first instructions of two successive iterations as adjacent, SLOTHY correctly takes into account the overlapping at the loop boundary.

Preamble and postamble. With software pipelining, the first and last iterations have to be treated especially: Concretely, a loop in which some instructions are being pulled into the previous iteration, requires a *preamble* consisting of the early instructions for the first iteration, and a *postamble*, consisting of the non-early instructions of the last iteration. Together, preamble and postamble then account for one full iteration. If both early and late instructions are used, preamble and postamble account for two full iterations. In either case, the loop counter needs to be adjusted accordingly.

A downside of the addition of preamble and postamble is an increased code-size. If this is problematic, an alternative is to replace the postamble by a full iteration of the optimized loop body, leaving only the (typically small) preamble as a code-size overhead. This approach is viable if (a) it is safe to read a fixed amount past the input buffer (e.g. because it is padded by the caller), and (b) the preamble does not include store instructions (which can be forced in the SLOTHY model). This approach is used e.g. in [Arme].

3.6 Performance/μArch constraints: Latencies

In this and the following section, we describe how we incorporate performance constraints into our model. For simplicity, we focus on the case of single-issue, in-order microarchitectures first. Superscalar, in-order microarchitectures are discussed in Section 3.9 below, and Section 3.10 comments on out-of-order microarchitectures.

In the context of a single-issue, in-order microarchitecture, an ideal, stall-free program execution would see the execution time difference (in cycles) between two instructions match their distance in program order. We therefore model latencies by requiring that for any $\texttt{I} \xrightarrow{\circ} \texttt{J}$, we have $\texttt{J.pos} \geq \texttt{I.pos} + \texttt{latency}(\texttt{I} \xrightarrow{\circ} \texttt{J})$. Note that this supersedes the previously introduced *functional* correctness constraint $\texttt{J.pos} \geq \texttt{I.pos}$. We also note that while $\texttt{latency}(\texttt{I} \xrightarrow{\circ} \texttt{J})$ often depends on I only, dedicated forwarding paths provide examples where $\texttt{latency}(\texttt{I} \xrightarrow{\circ} \texttt{J})$ may be smaller than the "generic" latency of I: On the Cortex-M55 CPU, for example, vector multiplication instructions typically have a latency of 2 cycles, yet sequences `VMULx; VSTRx` run stall-free.

3.7 Modelling stalls

Performance constraints such as the ones discussed in Section 3.6 may not be satisfiable: Sometimes, stalls are unavoidable. We model stalls by allowing the set of program order positions to have gaps, constraining pos to be an injective map $\{0, \ldots, |\texttt{code}|\} \to \{0, \ldots, |\texttt{code}| + \texttt{maxgaps}\}$ rather than a permutation on $\{0, \ldots, |\texttt{code}|\}$. The integer variable maxgaps models the maximum stall "allowance".

We consider two approaches for minimizing maxgaps: In the first approach, we fix maxgaps at model construction time. If the resulting model has a solution, we try again with a smaller value for maxgaps; otherwise, we increase it. Using binary search, a small number of iterations should reveal the minimum stall allowance. In the second approach, we model maxgaps as an undetermined integer variable that is part of the model, and mark its minimization as the *objective* of the underlying solver, if supported.

Our experiments with CP-SAT did not render either approach generally better than the other: For optimization problems of small or medium complexity, modelling stalls "internally" in the model appears to work well and lead to the optimal solution faster than an "external" binary search. For highly complex models, however, adding further dynamics through the flexible maxgaps could sometimes render the optimization problem infeasible (that is, we would abort the solver prior to having found a solution), while the solver could make progress using an "external" binary search.

3.8 Performance/μArch constraints: Instruction Overlapping

Next, we describe how we model functional units and instruction overlapping. As before, we make the simplifying assumption that the target microarchitecture is single-issue and in-order, leaving generalizations to Section 3.9 and Section 3.10 below.

Instruction overlapping is modeled by assigning to each instruction `I` a *functional unit* `Unit(I)` (depending on the target microarchitecture) and a usage time (in cycles) `block(I)` capturing for how long `I` keeps `Unit(I)` busy. We then demand that for any fixed functional unit `U`, the set of intervals $\{[\texttt{I.pos}, \texttt{I.pos} + \texttt{block(I)}) \mid \texttt{Unit(I)} = \texttt{U}\}_{\texttt{I}}$ is non-overlapping. This is a straightforward instance of the job shop problem. If an instruction may run on multiple functional units, we create Boolean variables tracking where they actually run, and condition the usage intervals accordingly — that is, we reduce to a *flexible* job shop problem. On the Cortex-M55 processor, an example for an instruction that can run on multiple functional units is the double vector-to-register move `VMOV Ra, Rb, Qn[i], Qn[j]`.

We highlight that the latency of an instruction may be smaller than its usage time: On Cortex-M55, for example, many vector instructions (e.g. `VADD`) occupy their functional units for 2 cycles, but have 1 cycle latency: This is possible since instruction overlapping supports data dependencies between the overlapping instructions.

3.9 Superscalar microarchitectures

We have so far described our model for *single-issue* microarchitectures, using a single integer variable `I.pos` for an instruction's position in program order as well as its (ideal) issue cycle. For in-order, *superscalar* microarchitectures, we need two separate, yet closely related variables: In addition to `I.pos`, keeping its meaning as the position of I in program order, we introduce another variable `I.cyc` for the issue cycle. If N is the maximum number of instructions issued per cycle, we relate `I.pos` and `I.cyc` via $\texttt{I.cyc} = \lfloor \frac{\texttt{I.pos}}{N} \rfloor$. Where such ideal issue-rate cannot be achieved, gaps will be left in the program order assignments, as described in Section 3.7. In particular, we note that N may be strictly larger than the issue-rate of the target microarchitecture — the true limit to multi-issuing is enforced through the configuration of functional units (Section 3.8) and microarchitecture-specific constraints capturing which instructions may be issued in the same cycle.

3.10 Out-of-order microarchitectures

A typical out-of-order CPU performs register renaming and instruction reordering in *hardware*. Those also being the core tasks of SLOTHY, one might wonder whether using SLOTHY for out-of-order CPUs is fruitful. Somewhat surprisingly, we found (and show later) that SLOTHY can be useful for some out-of-order CPUs as well, as we explain now.

First, the amount of renaming and reordering a given out-of-order CPU is capable of is bounded, and a trade-off between performance and hardware cost: The number of physical registers available for renaming is fixed, as is the width of the reordering window and the size of the issue queues. We therefore expect that code which is scheduled in consideration of execution units and latencies, is less likely to hit the limits of the reordering capabilities, and should thus perform better. Concretely, for example, instructions scheduled in consideration of the latencies of their source instructions will have a shorter wait in the issue queue and are therefore less likely to lead to overflows in the latter.

Second, the out-of-order microarchitectures considered here have in-order *frontend* which takes care of, among other things, register renaming and instruction dispatch to issue queues. If the width of this frontend (for the instruction classes under consideration) is close to the width of the execution backend (for the same class), a constantly high throughput of the former is required for high utilization of the latter — we will look at

specific examples later. Moreover, the frontend being in-order, its throughput maximization becomes a scheduling sensitive problem that we can attempt to model in SLOTHY.

3.11 Input and output registers

As mentioned in Section 2.1, we model inputs and outputs as special nodes in the computational flow graph, with input nodes not having any inputs, and output nodes not having any outputs. Input nodes are added automatically during CFG construction: If an instruction takes an input argument which has not been written to so far, a new input node is created and a dependency added between that input node and the current instruction. Output nodes, in turn, are created on behalf of the user: SLOTHY cannot know which registers written to by the current program excerpt are actually relevant for the remainder of the program, and which ones are merely temporaries — it's for the user to say. For every output register, we then create an output node dependent on the last instruction that wrote the register. A noteworthy special case is when a register declared as an 'output' is never written to: In this case, a pair of depending input and output nodes is created for that register, thereby preventing other instructions from using the register.

During the construction of the constraint model, input and output nodes are largely treated like ordinary instruction nodes: They have a program order position, input and output dependencies, and contribute to the tracking of register lifetimes and usage. They are only special in that they have their position fixed to the beginning/end of the program, and do not enter *micro*architectural constraints. This uniformity simplifies the model significantly — an earlier version of SLOTHY did not use input and output nodes, and was considerably more complex and error-prone.

3.12 Memory, stack and flags

SLOTHY does not have an in-built notion of memory. In our Armv8.1-M+Helium and AArch64+Neon models discussed below, store instructions are modelled as having no output, while load instructions are modelled as having no input except for the address register. This approach is sound if there are no data dependencies through memory *within* the range of a single SLOTHY optimization, which is the case for the common load-operate-store style loop not relying on the stack as temporary storage.

Where use of the stack is required, the following simple trick can be used: Stack locations can be modelled as "registers" of a separate register type, which are written to and read from via virtual instructions that have the same properties as loads/stores (e.g. in terms of latencies and functional units used). With this approach, which fits into the existing framework of SLOTHY, data dependencies can be tracked across stack spills without having to model linear memory. This approach works more generally for any *static* use of memory. We use it to model the stack in some of the NTT examples as well as the X25519 example discussed in Section 6.4.

A similar trick applies to modelling arithmetic flags: Each such flag can be modelled as a "register" of a dedicated register type, and instructions are marked as depending on or modifying those registers in accordance with their reliance and influence on arithmetic flags. We use this approach to model flags in the X25519 example below.

3.13 Address modifications

A typical loop will modify the base address for the data to be operated on with each iteration: For example, in Helium, data might be loaded via `VLDR Qin, [Raddr]` for some address register `Raddr`, and later stored via `VSTR Qout, [Raddr], #16`, incrementing the address register for the next iteration. Such patterns make deep software pipelining impossible without further modelling of the semantics of the address

increments, as there is a data dependency between the final store of one iteration, incrementing the address register, and the initial load for the next iteration. Since SLOTHY does not know about the semantics of instructions beyond their signature and dependencies, it cannot reorder load and store in this situation. *With* further semantic knowledge, however, one *could* of course reorder load and store in such situations, leveraging commutativity relations such as `VSTR Qa, [Rptr], #16; VLDR Vb, [Rptr]` ≡ `VLDR Qb, [Rptr, #16]; VSTR Qa, [Rptr], #16`.

We address this problem as follows: To begin, our existing SLOTHY architecture models (see Section 4) do not model address increments, but treat address registers for load/store operations as input-only. This gives SLOTHY the flexibility to freely reorder load/stores and identify deep software pipelining opportunities, but also constitutes a semantic change if a load/store with increment is reordered with another load/store from the same base register. *After* the optimization, we therefore iterate through pairs of load/stores which depend on the same address register and have been reordered by SLOTHY, and fix up their address offsets, leveraging commutativity relations such as the one above. This works well in the typical case where only one load/store instruction modifies the base register, and where the offers a flexible immediate offset. For pairs of load/store instruction which do not have this flexibility, we forbid their reordering in SLOTHY.

We note that manual address increments via `add Raddr, Raddr, #16` do currently still prevent software pipelining in SLOTHY. For loops where such increments can not be realized as pre/post increments, further work on SLOTHY is needed to optimize them.

3.14 Other objectives

Beyond the minimization of stalls, the following are useful objectives: If software pipelining is enabled, it is natural to maximize $\sum_{\texttt{I}}$ `I.core` — that is, to minimize the amount of interleaving between successive iterations. Second, register usage can be minimized. This is particularly interesting if the code in question does not fit into the register file, and "virtual registers" are used to approximate the amount of stack spilling needed.

3.15 Performance fidelity

Ideally, one would like SLOTHY's view of some code's performance to be perfectly accurate. However, as we do not expect SLOTHY to be used with complete microarchitectural models, this is unlikely. Instead, one only models key aspects such as instruction latencies or throughput, and optimizes code according to those. Unmodeled microarchitectural aspects may therefore still lead to stalls in code which the (micro)architectural model underlying SLOTHY considers "stall-free". It is therefore important to validate the performance of code produced via SLOTHY by running it on real hardware.

3.16 Soundness & Trust

Soundness means that SLOTHY emits code which is functionally equivalent to the original code. Considering the complexity of the components that go into a SLOTHY-based optimization — including SLOTHY itself, its implementation, and the underlying constraint solver — it is important to have an approach to establishing soundness that does not require trust in either of those components. In this section, we sketch such an approach — full elaboration and mechanization, however, is left for future work.

The essential idea is that for optimizations that do not change instructions but only their order and register usage, correctness should be a consequence of the produced code having a computational flow graph (CFG) isomorphic to that of the input code. Moreover, such an isomorphism can be checked for efficiently once provided with the underlying permutation of the optimization — the "certificate"/witness of the isomorphism in the complexity

theoretic sense — which is part of the solver's output. In other words, a CFG-preserving optimization step can be verified independently of the constraint modelling approach, the implementation, and the underlying solver: Instead, only the (manual or automatic) code-to-CFG parser as well as the graph isomorphism checker would need to be trusted. Our SLOTHY implementation (see Section 5) itself performs such a CFG-isomorphism check before outputting results of core optimization steps, but independent verification tooling can and should be implemented.

There are some non-trivial details to be considered in the above blueprint, which we briefly outline. First, the approach assumes that the semantics of some code is a function of the code's CFG, which requires the CFG to e.g. at least include reads/writes to memory. However, as explained in Section 3.12, SLOTHY does not consider dependencies through memory and elides it from the CFG. This is a valid transformation under the assumption that the code under consideration does indeed not have dependencies through memory, but nonetheless constitutes a CFG-altering preprocessing prior to the core CFG-preserving optimization that needs independent verification. A similar comment applies to address modifications (Section 3.13), which too can be viewed as CFG-altering preprocessing steps.

In summary, we believe that SLOTHY-based optimization is amenable to automated certificate-based verification with SLOTHY and its implementation being untrusted. However, further work is required to elaborate and implement this verification strategy — our own implementation currently only performs the core CFG-isomorphism "selfcheck".

4 (Micro)Architecture models

Section 3 described our (micro)architecture-agnostic framework SLOTHY for assembly optimization via constraint solving. In this section, we discuss our concrete instantiations: We model aspects of the Cortex-M55 and Cortex-M85 CPUs implementing Armv8.1-M+Helium, as well as the Cortex-A55 and Cortex-A72 CPUs implementing AArch64+Neon.

4.1 General approach

Architecture modelling. A SLOTHY architecture model has the following main components: First, the set of register types and their associated registers. Register aliases (such as `LR` being an alias for `X30` in AArch64) may be defined, too. Second, the set of instructions required for the workload(s) to be optimized. For every instruction, SLOTHY needs to know about the number of input, output, and input/output registers and their register type. If there are restrictions on which registers can be used, those *must* be modelled, too. Jointly destructive instruction patterns *should* be identified and modelled for best performance. Finally, parsers and writers need to be written which SLOTHY can use when converting between assembly code and computational flow graphs. To reduce the complexity of parsers/writers, users may define assembly macros wrapping individual instructions into simpler syntax.

Microarchitecture modelling. A SLOTHY microarchitecture model has the following main components: First, the set of functional units and, for a superscalar architecture, the issue width. Second, the latencies, throughputs, and functional unit of every instruction. Third, special cases and target specific restrictions, a typical example being forwarding paths for common instruction sequences, or issue-slot restrictions.

For in-order microarchitectures, we extracted most of the required information from the respective software optimization guides (SWOGs), with additional validation and refinement using our own microbenchmarks.

For out-of-order microarchitectures, the modelling process appears less precise and more heuristic: We also gathered information regarding the throughput and latency from

the SWOGs, but experimented with relaxations on the latencies because of out-of-order execution, instead incorporating frontend limitations as outlined in Section 3.10.

4.2 Armv8.1-M+Helium

For an introduction to Armv8.1-M+Helium, we refer back to Section 2.2.2.

Cortex-M55. The Cortex-M55 software optimization guide [Armk] (SWOG) provides very detailed microarchitectural information that can converted into a precise SLOTHY model for the Cortex-M55 processor. The most important points are the following: First, Cortex-M55 is a dual-beat implementation of Helium, that is, every Helium instruction runs for two cycles. In terms of the SLOTHY model, this means that the throughput of all Helium instructions is modelled as 1/2. Second, there are separate Helium execution pipelines for (a) load/store, (b) integer, and (c) floating point / integer multiplication instructions, which are modelled as separate "execution units" in the sense of the SLOTHY model. Third, the SWOG details the latencies of Helium instructions, which are reflected in our SLOTHY model. We also include a latency exception for integer multiplications `VMULx`, which generally have a latency of 2 cycles, but forward to immediately following vector-store operations without stall.

Finally, we constrain the SLOTHY model for Cortex-M55 to forbid patterns of the form `VSTx; ?; VLDx`, as those patterns do sometimes, depending on address alignment, lead to memory bank conflicts, and thus pipeline stalls. Below, this phenomenon will be referred to as a *ST-LD-hazard* (the interested reader should consult the SWOG for detailed load/store-pipeline information explaining this).

Cortex-M85. We also experiment with a SLOTHY model for the Cortex-M85 CPU based on information received from Arm— a Cortex-M85 software optimization guide is not yet available. We focus on the microarchitectural aspects related to Helium only: Cortex-M85 brings significant scalar improvements compared to Cortex-M55 because of larger dual-issuing capabilities, but we leave their modelling and use for future work.

In terms of Helium, the Cortex-M85 microarchitecture is overall similar to Cortex-M55, with a few noteworthy differences: First, similar to Cortex-M55, there are separate pipelines for integer and floating-point addition/logical instructions, and for integer and floating-point multiply instructions. Second, loads and stores can overlap (so that e.g. `VLDRx; VSTRx; VLDRx; ...` can run stall-free), and only some specific sequences such as `{ARITH}; VLDx; VSTx; VLDx` can lead to stalls — overall, scheduling of loads and stores is more flexible. Third, the Cortex-M85 microarchitecture is designed to be implementable at higher clock-rates than Cortex-M55, which increases some (cycle) latencies compared to Cortex-M55: For example, de-interleaving loads `VLD2x` and floating-point MACs have a latency of 4 cycles on Cortex-M85. This reinforces the importance of good instruction scheduling, and also explains that, as we will see later (e.g. Section 5.3), the best implementations for Cortex-M55 and for Cortex-M85 may be different.

4.3 AArch64+Neon

Cortex-A55. We derived the SLOTHY model for Cortex-A55 from experimentation, the Cortex-A55 SWOG [Armi], and useful observations from [Len19]. Our goal was not to define a fully precise reflection of the Cortex-A55 microarchitecture, but to focus on — and model in *sufficient* detail — the aspects most relevant for our target workloads.

The Cortex-A55 is a superscalar CPU which can dual-issue many pairs of instructions including scalar and 64/128-bit Neon instructions, as well as pairs of 64-bit Neon instructions. Notable exceptions are most pairs of 128-bit Neon instructions, and pairs of multiply-accumulate scalar instructions. We express those characteristics in SLOTHY

as follows: First, we add *two* execution units for 64-bit vector operations, but mark 128-bit vector instructions as occupying both. Second, for scalar operations, we also model two execution units, and mark only one as capable of performing multiple-accumulate instructions; this is a slight deviation from the SWOG, which shows 3 scalar pipelines, but our model is sufficient for our purposes. Third, we model separate execution units for load and store operations, in accordance with the SWOG. Finally, we reflect issue-slot restrictions documented in the SWOG in our model, an important example being that most 128-bit Neon instructions are only issued on slot 0.

Latencies and throughputs are modelled as provided in the SWOG, with occasional changes as suggested by our microbenchmarks. We also model the forwarding paths for scalar and Neon multiply-accumulate sequences in SLOTHY, including their additional constraints identified and highlighted in [Len19].

Cortex-A72. In contrast to the Cortex-M55, Cortex-M85 and Cortex-A55 microarchitectures considered above, the Cortex-A72 is a triple-issue, *out-of-order* microarchitecture. We refer to Section 3.10 for the general discussion of the uses SLOTHY can have in this context. For Cortex-A72, our SLOTHY model captures a somewhat heuristic combination of frontend and backend constraints, as we explain now.

The first important observation is that the Cortex-A72 has two Neon execution pipelines in its out-of-order backend [Armj, Section 2.1], while the frontend can dispatch one micro-operation per cycle to the respective issue queues [Armj, Section 4.1]. This implies that Neon-heavy workloads, such as the Number Theoretic Transform (NTT), can achieve an optimum of 2 Neon Instructions per Cycle (IPC) only if the in-order frontend also achieves this throughput — a scheduling sensitive problem amenable to encoding in SLOTHY. Concretely, code should be scheduled so that in every cycle, exactly two Neon instructions are dispatched, one for each of the two Neon pipelines.

Motivated by the above, our SLOTHY-model of Cortex-A72 has as "execution units" the dispatch slots in the Cortex-A72 in-order *frontend* (detailed in [Armj, Section 4.1]). In this context, the notion of "latency" is not immediately meaningful since an instruction's readiness for dispatch is independent of its readiness for execution. Similarly, the dispatch throughput is usually different from the execution throughput; for example, 128-bit Neon integer multiplication instructions can be dispatched at a rate of 1 IPC, even though their execution happens with at most $1/2$ IPC on Cortex-A72. Despite this semantic discrepancy, we found that modelling the frontend latency and throughput constraints in accordance with the backend constraints, lead to good results. Intuitively, this is plausible, since a dispatch rate that's consistently higher than the backend's execution rate would eventually overflow the issue queues and lead to stall.

4.4 Other microarchitectures

Finally, a few words on the suitability of SLOTHY for other microarchitectures.

M-Profile. Within the space of M-profile CPUs, the Cortex-M4 has received much attention in the context of optimized PQC implementations [KPR+]. However, we do not see it as particularly promising target for the use of SLOTHY, as the Cortex-M4 is a non-superscalar microarchitecture with the majority of the instructions completing in a single cycle — scheduling of instruction is therefore largely irrelevant (with some exceptions, such as loads and stores [Armc]). Similar remarks apply to most of the more efficiency-focused M-profile CPUs. The Cortex-M7 CPU, in turn, could be an interesting target for future work, as it offers a 6-stage superscalar pipeline with the support for dual-issuing, as well as two ALUs with one supporting SIMD instructions [Armd]. Finally,

SLOTHY could be used as a vehicle to evaluate the performance impact of candidates for Arm Custom Instruction (ACI).

A-Profile. Within the spectrum of A-profile CPUs, we did not cover the Cortex-X series of performance-centric, highly out-of-order microarchitectures. Somewhat surprisingly, we believe that those high-end processors should also be amenable to optimization through SLOTHY for Neon-heavy workloads: Taking Cortex-X1 as an example, it can be seen from the software optimization guide [Arml] that there are 4 Neon pipelines in the backend, while the frontend also has a maximum throughput of 4 Neon-IPC. We are therefore in the situation, outlined in Section 3.10, where the theoretically optimal throughput seems achievable only with careful instruction scheduling in consideration of the frontend's dispatch constraints. We consider this an interesting research avenue to explore.

Finally, there are newer generations of A-profile cores in the Cortex-A5x and Cortex-A7x lines one can consider, such as the Cortex-A510 processor and the Cortex-A78, Cortex-A710 and Cortex-A715 processors. We believe that SLOTHY will generally prove useful for the Cortex-A5x in-order range — the Cortex-A510 CPU is particularly interesting because of its configuration, sharing two Neon-units between two CPUs. Starting from the Cortex-A78 CPU, however, the newer generations of Cortex-A7x cores seem the least likely to benefit from SLOTHY for Neon workloads: While they have a backend throughput of 2 Neon-IPC, the frontend can still achieve a maximum of 4 Neon-IPC (like the high-end), and so we expect to observe less scheduling sensitivity. Where newer generations of A-profile cores are explored, incorporating new vector extensions such as SVE and SVE2 into the SLOTHY (micro)architecture models should be considered as well.

5 SLOTHY as a development tool

In this section, we describe our implementation and the intended usage of SLOTHY.

5.1 Overview

Goal. We aim for SLOTHY to enable an automated, non-interactive and auditable optimization-step, operating on a clean and readable "base implementation" maintained by the developer(s). In this vision, the developer's responsibility and focus is to identify algorithms for the target workload and to express those in terms of the target architecture, without being impeded by microarchitectural performance concerns. The base implementation ought to be auditable and verifiable because of its clarity, while the SLOTHY-generated optimizations ought to be auditable and eventually verifiable on the basis of information emitted by SLOTHY, *without* trust in SLOTHY or its implementation.

We emphasize that "clean assembly" is not an oxymoron: `.req` directives allow for the use of symbolic register names, while `.macro` directives allow for the encapsulation of common functionality. The examples discussed below will demonstrate that those two constructs alone enable very readable assembly implementations.

Dependence on microarchitecture. Sometimes, there are multiple approaches to expressing a workload in terms of the target architecture's instruction set, with the best choice depending on the target *micro*architecture. We will see a few examples later. In those instances, we expect developers to maintain multiple base implementations reflecting the different approaches. Then, for a given microarchitecture target, `slothy` would be run on *all* base implementations, and the most suitable optimization chosen for use.

Compiler-based approaches. Using SLOTHY frees the developer from per-workload microarchitecture-specific optimizations. However, it remains a more laborious and less convenient development approach than the use of increasingly powerful compiler-based techniques such as intrinsics and or even auto-vectorization. We primarily see SLOTHY as worthwhile in contexts where unlocking the last % of performance is desirable, and/or where high workload complexity lowers the efficacy of compilation-based techniques. Further, we believe that SLOTHY can be a useful tool for the discovery of new optimization potentials, supporting compiler development.

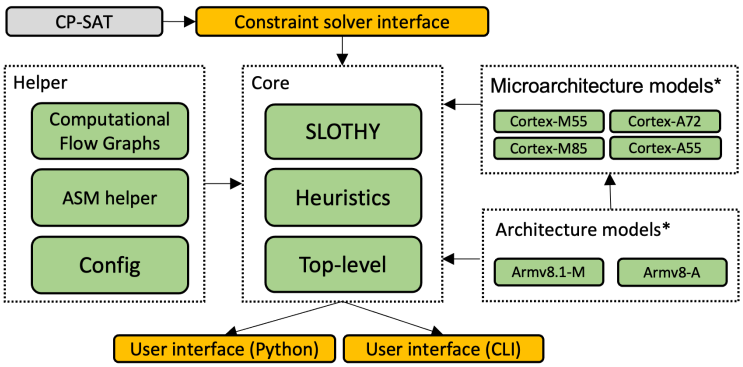

Figure 6: High-level structure of `slothy`, our implementation of SLOTHY.

5.2 Implementation

We implement the SLOTHY framework in Python, referred to as `slothy` in the following.

User interface. `slothy` can be called from Python or via the command line application `slothy-cli`. Convenience wrappers are provided for automatic instantiation to our various target (micro)architectures.

Constraint solver interface. For the interface to the underlying constraint solver, a small abstraction layer is introduced, facilitating the experimentation with different constraint solvers. Currently, `slothy` is based on CP-SAT from Google OR-Tools.

Implementation structure. Figure 6 provides an overview over the different components of `slothy`. The core implements the SLOTHY constraint modelling approach and provides some heuristics expanding the reach of `slothy` to larger kernels. Other helper modules provide configurability, functionality around computational flow graphs, and assembly support functions (such as support for `.req` and `.macro`). Finally, as discussed, there are multiple architecture and microarchitecture modules describing the target of optimizations.

Usage. After fixing a target architecture and microarchitecture, the user loads an assembly file, which can range from a small stub to a standalone assembly file with multiple functions, macros, or similar — `slothy` does not attempt to make sense of it at this point. The user then provides a configuration and issues one or more optimization commands, operating on selected parts of the loaded file in-place. Finally, functions may be renamed to avoid symbol clashes, and the optimized version of the code be stored to a file.

Optimizations are typically focused on a particular region, specified by assembly labels. In case of a loop, the label at the start of the loop body is sufficient. `slothy` will only attempt to parse the instruction in the selected regions. In particular, even when

```
  .syntax unified
  .global cmplx_mag_sqr_fx
  .type   cmplx_mag_sqr_fx, %function

  // Aliases for readability
  out   .req r0
  in    .req r1
  size  .req r2

  .text
cmplx_mag_sqr_fx:
  push  {lr}
  vpush {d8-d15}

  lsr lr, size, #2
  wls lr, lr, end

  .p2align 2 // Ensure loop alignment
start:
  // deinterleave real/imag
  vld20.32        {qreal, qimag}, [in]
  vld21.32        {qreal, qimag}, [in]!
  // square real/imag components
  vmulh.s32       qtmp, qreal, qreal
  vmulh.s32       qout, qimag, qimag
  // accumulate & halving
  vhadd.s32       qout, qout, qtmp
  // store result
  vstrw.32        qout, [out], #16
  le              lr, start
end:
  vpop {d8-d15}
  pop {lr}
```

Listing 3: "Base" implementation of vectorized squared magnitude function in Helium

standalone files are loaded, the underlying (micro)architecture models need only specify the regions to be optimized, allowing example-driven expansion of the models.

Symbolic registers. `slothy` supports the use of assembly "templates" using *symbolic* registers which have not been concretized using `.req` aliases. In this case, `slothy` takes care of finding suitable instantiations. This further simplifies the provision of the "base" implementation as it frees the developer from having to assign concrete architectural registers to logical variables. In many cases, `slothy` can instantiate the template without any reordering of instructions or renaming of registers which are already architectural — this is desirable as it allows the base implementation to be used as a functional reference. In more complex cases, the instantiation of symbolics has to be conducted in conjunction with optimization, as the template version of the code might not have sufficient free registers.

5.3 Toy example

We illustrate the intended usage of `slothy` by optimizing a simple Digital Signal Processing (DSP) kernel on the Armv8.1-M+Helium Cortex-M55 and Cortex-M85 CPUs.

The workload. We are optimizing a vectorized *squared magnitude* `sqmag` : $(z_1, \ldots, z_n) \mapsto (|z_1|^2, \ldots, |z_n|^2), z_i \in \mathbb{C}$. Specifically, we consider an implementation in 32-bit *fixed point* arithmetic, where the real and imaginary components of the z_i are represented in the Q1.31 fixed point format and where the outputs $|z_i|^2$ are stored in Q3.29.

The base implementation. Listing 3 shows a straightforward and largely self-explanatory implementation of `sqmag` in Armv8.1-M, using symbolic registers during the core computation (the allocation of the input arguments is fixed as part of the calling convention and must not be left to `slothy`). While readable, however, we note that the code is poorly scheduled from the perspective of instruction overlapping (recall Section 2.2.2).

Performance expectations. Each iteration performs 4 squared-magnitude operations $z \mapsto |z|^2$ via 6 Helium instructions. Moreover, 3 of those instructions are arithmetic, while the other 3 are load/store, which suggests a theoretical optimum of 1.5 cycles per squared magnitude operation on dual-beat CPUs like Cortex-M55 and Cortex-M85 — *provided* instruction scheduling is done carefully to achieve good instruction overlapping.

```
> for uarch in M55 M85; do for i in 1 2 4; do
   slothy-cli Arm_v81M Arm_Cortex_${uarch} examples/naive/cmplx_mag_sqr/cmplx_mag_sqr_fx.s \
    -l start -c sw_pipelining.enabled=True -c sw_pipelining.unroll=${i} -c timeout=5        \
    -r cmplx_mag_sqr_fx,cmplx_mag_sqr_fx_opt_${uarch}_unroll${i}                             \
    -o cmplx_mag_sqr_fx_opt_${uarch}_unroll${i}.s ;
  done; done
```

Listing 4: `slothy` command line for the optimization of `sqmag` "base" implementation for Cortex-M55 and Cortex-M85, with different levels of unrolling.

```
  .syntax unified
  .type
     cmplx_mag_sqr_fx_opt_M55_unroll2, %
     function
  .global
     cmplx_mag_sqr_fx_opt_M55_unroll2

  // ... aliases ...
  // assumes: sz >= 8, multiple of 4

  .text
  .align 4
cmplx_mag_sqr_fx_opt_M55_unroll2:
  push {r4-r12,lr}
  vpush {d0-d15}

  lsr lr, sz, #2
  wls lr, lr, end
  vld20.32 {q2,q3}, [r1]  // *......
  // gap                  // .......
  vld21.32 {q2,q3}, [r1]! // .*.....
  // gap                  // .......
  vld20.32 {q4,q5}, [r1]  // ..*....
  vmulh.s32 q0, q2, q2    // ...*...
  vld21.32 {q4,q5}, [r1]! // ....*..
  // gap                  // .......
  vmulh.s32 q7, q4, q4    // .....*.
  // gap                  // .......
  vmulh.s32 q4, q3, q3    // ......*
  lsr lr, lr, #1
  sub lr, lr, #1
.p2align 2
start:
  vld20.32 {q2,q3}, [r1]   // e..........
  vmulh.s32 q6, q5, q5     // .........*..
  vld21.32 {q2,q3}, [r1]!  // .e.........
  vhadd.s32 q1, q4, q0     // ....*.......
  vld20.32 {q4,q5}, [r1]   // ......e.....
  vmulh.s32 q0, q2, q2     // ..e.........
  vld21.32 {q4,q5}, [r1]!  // .......e....
  vhadd.s32 q2, q6, q7     // ..........*.
  vstrw.u32 q1, [r0] , #16 // .....*......
  vmulh.s32 q7, q4, q4     // ........e...
  vstrw.u32 q2, [r0] , #16 // ..........*
  vmulh.s32 q4, q3, q3     // ...e........
  // gap                   // ............

  le lr, start
  vhadd.s32 q6, q4, q0     // .*...
  vmulh.s32 q2, q5, q5     // *....
  vstrw.u32 q6, [r0] , #16 // ...*.
  vhadd.s32 q6, q2, q7     // ..*..
  vstrw.u32 q6, [r0] , #16 // ....*
end:
  vpop {d0-d15}
  pop {r4-r12,lr}
  bx lr
```

Listing 5: `sqmag` from Listing 3 optimized for Cortex-M55 by `slothy`.

Optimization. Listing 4 shows the command line optimizing our `sqmag` base implementation for Cortex-M55 and Cortex-M85. After choosing target architecture and microarchitecture, we enable software pipelining and try different levels of unrolling. Finally, we rename function (`-r`) and file (`-o`) and store the result.

Output. Listing 5 shows the file emitted by `slothy` in the case of the Cortex-M55 optimization of the doubly-unrolled code. We now explain the different components.

First, we see that the symbolic registers `qreal`, `qimag`, `qtmp`, `qout` have been replaced by concrete architectural registers. Next, the loop body has doubled in size due to the unrolling configuration `-c sw_pipelining.unroll=2`, and, accordingly, the loop counter is halved (`lsr, lr, lr, #1`). Crucially, the loop body has been renamed, reordered, and iterations interleaved: Comments indicate the reordering, with `e` annotating "early" instructions pulled forward from the next iteration. As a result, the first and last iteration had to be treated especially, explaining the newly added loop preamble and postamble as well as the loop counter correction `sub lr, lr, #1`. The `// gap` annotation indicates where `slothy` expects a stall, here due to a potential `ST-LD` hazard.

The permutation indicated in the comments forms the main "certificate" of the optimization: SLOTHY itself checks that it yields an isomorphism between input and output CFGs, but external tooling could mechanically verify this as well (based on a more suitable output format), illustrating the general trust story outlined in Section 3.16.

	Opt. for Cortex-M55			Base	Opt. for Cortex-M85		
Unrolling	4×	2×	1×	1×	1×	2×	4×
Cortex-M55	1.56	1.62	2.0	3.25	2.0	1.75	1.87
Cortex-M85	1.68	1.75	2.25	3.0	2.25	1.5	1.5

Table 1: Cycles / squared magnitude operation $z \mapsto |z|^2$ of base and `slothy`-optimized implementations. Horizontal: Optimization target & Unrolling level. Vertical: Benchmarking target.

Results. Table 1 shows the average cycle counts per squared-magnitude operation for the base and each of the unrolled & auto-optimized versions of `sqmag`, benchmarked on Cortex-M55 and Cortex-M85. We highlight the following: First, while readable, the naïve base implementation falls short of the optimal performance by 2×, underlining the need for optimization. Second, on Cortex-M55 we get closer to the theoretical optimum of 1.5 cycles with increasing level of unrolling, but we do not reach it (it is easy to argue theoretically why a stall-free version does indeed not exist). On Cortex-M85, in turn, we achieve perfect performance of 1.5 cycles from unrolling level 2 onwards. Third, the fastest code on Cortex-M55 is not the fastest on Cortex-M85, and vice versa, underlining the benefit of the automated microarchitecture-specific optimization provided by `slothy`.

Reflection. While simple, the example of `sqmag` demonstrates the capabilities and our envisioned usage of `slothy` well: We could focus on mapping `sqmag` to Armv8.1-M+Helium in a readable way, leaving all microarchitecture specific optimizations — which, even in this short kernel, are not trivial — to `slothy`, taking the target microarchitecture as a parameter. Of course, we had to develop and a suitable microarchitecture model for `slothy`, but this is one-off effort that can be re-used for other workloads.

5.4 Performance

Judging from numerous examples, we find that the performance of `slothy` depends on the length of the kernel to be optimized, the complexity of the microarchitectural model, and finally — in an intuitive sense — the difficulty of the optimization problem. Moreover, due to the non-deterministic and heuristic nature of CP-SAT, the runtime and results of `slothy` varies even for the same workload. In fact, the runtime could even depend on the precise order in which the model's variables and constraints are presented to the solver, but we leave exploration and utilization of this variability for future work.

From Figure 7 we can see that kernels of less than 50 instructions typically optimize in seconds to minutes while kernels between $50 - 250$ instructions tend to optimize in minutes to hours. We sometimes also note a large performance variability for the same kernel, again owing to the non-determinism of CP-SAT. Further, the intuition that more complex microarchitectural models should increase the runtime of `slothy` is also reflected in Figure 7: For two data points of different targets that have the same number of instructions (and thus correspond to the same code snippet, most of the time), we can see that, in the majority of cases, the optimization on the Cortex-A72 (resp. Cortex-M85) takes more time than on the Cortex-A55 (resp. Cortex-M55).

Starting from about 150 instructions, optimizing some kernels with our AArch64+Neon model tends to be infeasible through the vanilla SLOTHY constraint modelling approach, while for our Armv8.1-M+Helium models, the threshold is at about 80 instructions. To still handle those — the X25519 example discussed below features a loop kernel of nearly 1000 instructions — we implement a series of heuristics, as discussed next.

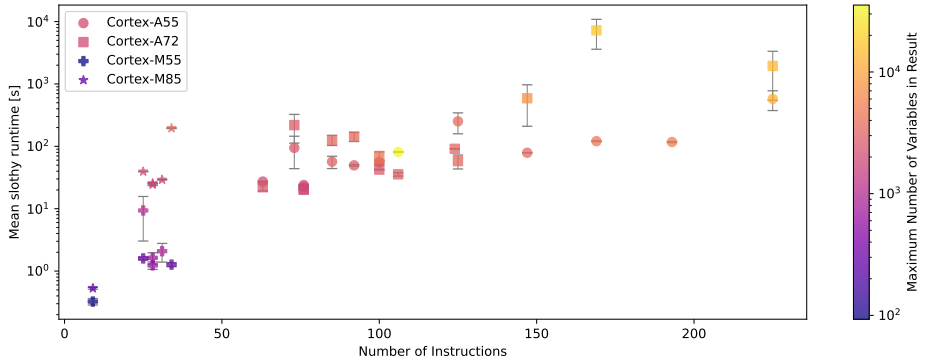

Figure 7: Runtime measurements of `slothy` for loops of various NTTs. One data point represents the runtime distribution of 10 optimizations of a fixed loop body, with software pipelining enabled. Optimization of pre/postamble disabled, no additional optimization targets set. Measurements done on Apple M1 Pro (8C).

5.5 Heuristics

In order to deal with code too large to be optimized via the vanilla SLOTHY approach, we implement a few simple heuristics. Those will mainly be used for X25519 discussed in Section 6.4, which involves a loop body of 958 instructions. We note, however, then when heuristics are used, (global) optimality of the optimized code is no longer guaranteed.

Splitting heuristic. A straightforward way to handle large code is to split it into multiple segments and optimize them individually. We added this functionality to SLOTHY and call it the *splitting heuristic*. It can be enabled using the `split_heuristic=True` argument and the number of segments configured via `split_heuristic_factor=N`.

To prevent instructions being trapped inside their segments, a sliding window can be used, with segment overlapping controlled by `split_heuristic_stepsize=frac`. For example, a factor of 5 and a stepsize of 0.1 results in separate optimizations for the segments $[i/10, i+2/10], i = 0, \ldots, 8$. This allows for some movement of instructions between segments, which can be amplified through repetition, controlled via `split_heuristic_repeat=N`.

Migration of instructions between segments can be further "encouraged" using the options `move_stalls_to_top/bottom`: Once the minimum number of stalls is found, this re-optimizes the code with the objective of moving the stalls to the top or bottom.

The option `split_heuristic_region` can be used to limit the optimization to a certain region inside the code.

Preprocessing. While functional in principle, the splitting heuristic is inefficient in situations where code-paths to be interleaved are very far — multiple segments — apart from each other. In such cases, simple CFG-based preprocessing can be used to establish coarse-grained interleaving prior to more refined optimization: For example, the option `split_heuristic_preprocess_naive_interleaving` interleaves instructions based on their depths, with instructions of lowest depths receiving highest priority. While the interleaving capabilities of this trivial approach are limited by the lack of register renaming (recall Listing 1), it proved effective in some examples.

(Periodic) Halving heuristic. For instances in which optimizing a loop kernel with `sw_pipelining.enabled` is computationally too complex, but some interleaving between iterations is still desired, we use the "halving heuristic": This heuristic first optimizes

```
    vldrw.32      q1, [in0]
    vldrw.32      q6, [in2]
2:
    vhadd.s32     q0, q1, q6
    vldrw.32      q4, [in1]          //?
    vhsub.s32     q2, q1, q6
    vldrw.32      q5, [in3]
    vhadd.s32     q1, q4, q5
    vhsub.s32     q3, q4, q5         //-
    vldrw.32      q7, [t1], #16
    vhadd.s32     q4, q0, q1
    vstrw.32      q4, [in0], #16
    vhsub.s32     q4, q0, q1
    vldrw.32      q5, [t0], #16      //?
    vqdmlsdh.s32  q0, q4, q5
    vhcadd.s32    q6, q2, q3, #270
    vqdmladhx.s32 q0, q4, q5
    vstrw.32      q0, [in1], #16
    vqdmlsdh.s32  q0, q6, q7
    vldrw.32      q1, [in0]          //?
    vqdmladhx.s32 q0, q6, q7
    vstrw.32      q0, [in2], #16
    vhcadd.s32    q4, q2, q3, #90
    vldrw.32      q5, [t2], #16      //?
    vqdmlsdh.s32  q0, q4, q5
    vldrw.32      q6, [in2]
    vqdmladhx.s32 q0, q4, q5
    vstrw.32      q0, [in3], #16
    le            lr, 2b
```

Listing 6: Handwritten implementation of a radix-4 layer of a fixed-point FFT from github.com/ARM-Software/EndpointAI.

the body of the loop with no software pipelining enabled. It then splits the optimized body into two halves $[a; b]$ and optimizes $[b; a]$, where b is the late half of one iteration and a the early half of the succeeding iteration. The final code contains a as its preamble, $\text{opt}([b; a])$ as the loop body, and b as the postamble. If we want SLOTHY to consider the seam between iterations, `halving_heuristic_periodic` can be set, which will enable software pipelining for $\text{opt}([b; a])$ but without allowing for early or late instructions.

6 Real world examples

In this section, we demonstrate the power of `slothy` by optimizing various real-world workloads: The FFT, the NTT, and the X25519 key exchange. First, however, we summarize some lessons learned from working through the examples to follow.

Effort. We found that the biggest effort in using SLOTHY is the *one-off* development of (micro)architecture models, *not* the development of the example source code: Since the latter can be written "cleanly" using macros and symbolic registers — that's the point of SLOTHY— workloads are typically quick to implement. For the architecture models, in turn, there is a large effort to be made to write parsers for all required instructions. Similarly, microarchitecture models take considerably time to refine to the point where even details like forwarding paths or latency exceptions are correctly encoded.

Pitfalls. The CFG isomorphism self-check within `slothy` guards against a lot of potential issues with SLOTHY and `slothy`, but some usage pitfalls remain that we ran into a few times: First, teething issues in the architecture model. For example, if one incorrectly models an input/output argument as an output argument, SLOTHY would perform invalid register renaming, yet it has no chance to detect so because the architecture model is, and has to be, trusted. Second, configuration issues in the *use* of `slothy`: A common pitfall was a lack of declarations of reserved registers that are unused by the snippet under consideration, and yet have to be preserved for the sake of the surrounding code.

6.1 Example: FFT on Armv8.1-M+Helium

In this section, we use `slothy` to optimize the 32-bit Fast Fourier Transform (FFT) on Armv8.1-M+Helium, targeting the Cortex-M55 and Cortex-M85 processors. We consider implementations in both fixed-point and floating-point format, and focus on a single radix-4 layer. Recall Section 2.4 for background on the FFT.

```
.macro load_data
  vldrw.s32 qA, [inA]
  vldrw.s32 qB, [inB]
  vldrw.s32 qC, [inC]
  vldrw.s32 qD, [inD]
.endm

.macro load_twiddles
  vldrw.s32 qTw1, [pW1], #16
  vldrw.s32 qTw2, [pW2], #16
  vldrw.s32 qTw3, [pW3], #16
.endm

.macro store_data
  vstrw.32 qA, [inA], #16
  vstrw.32 qB, [inB], #16
  vstrw.32 qC, [inC], #16
  vstrw.32 qD, [inD], #16
.endm
```

```
.macro cmul_fx out, in0, in1
  vqdmlsdh.s32  \out,\in0,\in1
  vqdmladhx.s32 \out,\in0,\in1
.endm

fx_rad4_loop_start:
  load_data
  load_twiddles
  vhadd.s32  qSm0, qA,   qC          // a+c
  vhadd.s32  qSm1, qB,   qD          // b+d
  vhsub.s32  qDf0, qA,   qC          // a-c
  vhsub.s32  qDf1, qB,   qD          // b-d
  vhadd.s32  qA,   qSm0, qSm1        // a+b+c+d
  vhsub.s32  qBp,  qSm0, qSm1        // a-b+c-d
  vhcadd.s32 qCp,  qDf0, qDf1, #270 // a-ib-c+id
  vhcadd.s32 qDp,  qDf0, qDf1, #90  // a+ib-c-id
  cmul_fx    qB,   qTw1, qBp         // Tw1*(a-b+c-d)
  cmul_fx    qC,   qTw2, qCp         // Tw2*(a-ib-c+id)
  cmul_fx    qD,   qTw3, qDp         // Tw3*(a+ib-c-id)
  store_data
  le lr, fx_rad4_loop_start
```

```
.macro cmul_flt out, in0, in1
  vcmul.f32  \out, \in0, \in1, #0
  vcmla.f32  \out, \in0, \in1, #270
.endm

flt_rad4_loop_start:
  load_data
  load_twiddles
  vadd.f32  qSm0,  qA,   qC
  vadd.f32  qSm1,  qB,   qD
  vsub.f32  qDf0,  qA,   qC
  vsub.f32  qDf1,  qB,   qD
  vadd.f32  qA,    qSm0,  qSm1
  vsub.f32  qBp,   qSm0,  qSm1
  vcadd.f32 qCp,   qDf0, qDf1, #270
  vcadd.f32 qDp,   qDf0, qDf1, #90
  cmul_flt  qB,    qTw1, qBp
  cmul_flt  qC,    qTw2, qCp
  cmul_flt  qD,    qTw3, qDp
  store_data
  le lr, flt_rad4_loop_start
```

Listing 7: Base implementation of radix-4 layer of a fixed-point and floating-point FFT, using symbolic registers and macros for readability.

```
vldrw.s32 q0, [inB]           //*.......
// gap                        //.......*.
vldrw.s32 q2, [inD]           //....*....
vhadd.s32 q1, q0, q2          //......*..
vldrw.s32 q7, [inA]           //..*......
// gap                        //........*
vldrw.s32 q4, [inC]           //.*.......
vhadd.s32 q6, q7, q4          //...*.....
vldrw.s32 q5, [pW2] , #16     //.....*...
loop_start:
  vhadd.s32 q3, q6, q1        //........*.............
  vstrw.u32 q3, [inA] , #16   //.........*............
  vhsub.s32 q6, q6, q1        //..........*...........
  vqdmladhx.s32 q3, q5, q6    //...........*..........
  vldrw.s32 q1, [pW1] , #16   //................*.....
  vqdmlsdh.s32 q3, q5, q6     //............*.........
  vldrw.s32 q5, [pW3] , #16   //....................*...
  vhsub.s32 q6, q0, q2        //.......*.............
```

```
vldrw.s32 q0, [inB, #16]    //..e.....................
vhsub.s32 q2, q7, q4        //.....*..................
vstrw.u32 q3, [inB] , #16   //..............*.........
vhcadd.s32 q7, q2, q6, #270 //...............*........
vqdmladhx.s32 q3, q1, q7    //.................*......
vldrw.s32 q4, [inC, #16]    //.e......................
vqdmlsdh.s32 q3, q1, q7     //..................*.....
vldrw.s32 q7, [inA]         //e.......................
vhcadd.s32 q1, q2, q6, #90  //...................*....
vstrw.u32 q3, [inC] , #16   //..................*.....
vqdmladhx.s32 q3, q5, q1    //....................*...
vhadd.s32 q6, q7, q4        //....e...................
vldrw.s32 q2, [inD, #16]    //...e....................
vqdmlsdh.s32 q3, q5, q1     //....................*.
vldrw.s32 q5, [pW2] , #16   //...........e............
vhadd.s32 q1, q0, q2        //......e.................
vstrw.u32 q3, [inD] , #16   //.....................*
le lr, loop_start
loop_end:
...
```

Listing 8: `slothy` optimization of Listing 7 for radix-4 fixed-point FFT. No stalls remain on Cortex-M55 for the core of the loop. Postamble omitted.

Prior implementations. The state-of-the-art FFT implementations for Armv8.1-M+Helium are from [Armh, Arme] and based on intrinsics and handwritten assembly. Exact references to the implementations we measured are given in Table 2. For example, Listing 6 shows a handwritten fixed-point radix-4 layer from [Arme].

Performance expectations. Unsurprisingly for a workload as important as the FFT, the existing implementations already achieve good performance, yet some stalls remain: For example, the handwritten assembly in Listing 6 has one structural hazard (`// -`) and four alignment-dependent `ST-LD` hazards (`// ?`) remaining. The performance of all our reference implementations is part of Table 2. Our goal is to use `slothy` to either confirm that the remaining stalls are inherent, or find that they can be further reduced.

Challenges. The floating-point FFT is more challenging to optimize than the fixed-point FFT, both on the Cortex-M55 and the Cortex-M85 processor: On Cortex-M55, floating point addition and multiplication operations do not overlap as they run on the same execution pipe. On Cortex-M85, they can overlap, but floating point multiply-accumulate operations have a latency of 4 cycles. This reinforces the importance of precise scheduling, and also suggests that the implementations on Cortex-M55 and Cortex-M85 may be meaningfully different.

	Type	Code	$\frac{\text{Cycles}}{\text{Butterfly}}$	R	F	Code	$\frac{\text{Cycles}}{\text{Butterfly}}$	R	F
CFFT Q.31			Cortex-M55				Cortex-M85		
	Intrinsics	[Armr]	30 (+16%)	✓	✓	[Armr]	29 (+13%)	✓	✓
	Handwritten	[Armq]	28 (+10%)	✗	✗	[Armq]	26 (+3%)	✗	✗
	`slothy`	Our work	25	✓	✓	Our work	25	✓	✓
CFFT FP32			Cortex-M55				Cortex-M85		
	Intrinsics	[Armo]	33 (+15%)	✓	✓	[Armo]	34 (+20%)	✓	✓
	Handwritten	[Armm]	29 (+3%)	✗	✗	[Armn]	29 (+6%)	✗	✗
	`slothy`	Our work	28	✓	✓	Our work	27	✓	✓

Table 2: Comparison of various FFT implementations for Armv8.1-M+Helium in fixed-point and floating-point arithmetic. Abbreviations: R=Readable, F=Flexible μArch.

Base implementation. Listing 7 shows our clean base implementations for the floating-point and the fixed-point FFT, using symbolic registers and assembler macros. We use more temporaries than available architectural registers, and leave it to `slothy` to reorder and rename instructions and registers to fit in the architectural register file.

Optimized implementations. To get a sense of the `slothy`-optimizations, Listing 8 shows the optimized fixed-point FFT for the Cortex-M55 CPU: In each iteration, *seven* instructions are pulled deep into the previous iteration, enabling a *stall-free* loop body — identifying such optimizations by hand would be very time-consuming. This version is also stall-free on Cortex-M85 — in fact, running `slothy` for Cortex-M85 finds a stall-free version with only *one* early instruction, owing to the simpler scheduling of load/stores.

Benchmarking environment. We benchmark all code using the MPS3 FPGA prototyping board and the freely available AN552 and AN555 application nodes for Cortex-M55 and Cortex-M85, respectively. Intrinsics have been compiled with Arm Compiler 6.19.

Results. Table 2 shows the performance of our `slothy`-optimized FFTs in comparison with the reference implementations. We improve upon the state of the art in *all* cases, achieving gains of $3\% - 10\%$ compared to the prior microarchitecture-specific handwritten assembly from [Arme], and of $13\% - 20\%$ compared to the intrinsics versions from [Armh]. Given that the FFT is an exceptionally important and very well-studied workload for DSP, we consider these improvements proof of the capabilities of `slothy`. Our code is readable and the approach adaptable to other microarchitectures, similar to intrinsics-based code.

Impact. Our optimized FFTs have been communicated to Arm and merged into [Arme].

6.2 Example: NTT on Armv8.1-M+Helium

We now present our first cryptographic example for the use of SLOTHY: The SLOTHY-based optimizations for the Number Theoretic Transform (NTT) underlying Kyber and Dilithium, targeting the Cortex-M55 and Cortex-M85 CPUs implementing Armv8.1-M+Helium.

6.2.1 Previous implementations

[BHK+21] explores how to map modular arithmetic in the NTT to the Arm architecture, including Armv8.1-M+Helium. [BBMK+21, BHK+22] leverage those primitives for complete implementations of 32-bit NTTs in Helium. These implementations keep the data in

```
// Barrett multiplication
.macro mulmod dst, src, c, c_twist
  vmul.s32      \dst,  \src, \c
  vqrdmulh.s32 \src,  \src, \c_twist
  vmla.s32      \dst,  \src, q
.endm
// Cooley-Tukey butterfly
.macro ct_butterfly a, b, r, r_twist
  mulmod    tmp, \b, \r, \r_twist
  vsub.u32  \b,  \a, tmp
  vadd.u32  \a,  \a, tmp
.endm
...
  ldrd r0, r0_twist, [r_ptr], #+8
  ldrd r1, r1_twist, [r_ptr], #+8
  ldrd r2, r2_twist, [r_ptr], #+8
```

```
layer12_loop:
  vldrw.u32 d0, [in_lo]
  vldrw.u32 d1, [in_lo, #256]
  vldrw.u32 d2, [in_hi]
  vldrw.u32 d3, [in_hi, #256]
  ct_butterfly d0,d2,r0,r0_twist
  ct_butterfly d1,d3,r0,r0_twist
  ct_butterfly d0,d1,r1,r1_twist
  ct_butterfly d2,d3,r2,r2_twist
  vstrw.u32 d0, [in_lo],#16
  vstrw.u32 d1, [in_lo, #-240]
  vstrw.u32 d2, [in_hi],#16
  vstrw.u32 d3, [in_hi, #-240]
  le lr, layer12_loop
layer12_loop_end:
```

Listing 9: Naïve implementation of two merged layers of a 32-bit, radix-2 Number Theoretic Transform, using macros for readability.

```
layer45_loop:
  load_next_roots
  vldrw.u32 d0, [in]
  vldrw.u32 d1, [in, #16]
  vldrw.u32 d2, [in, #32]
  vldrw.u32 d3, [in, #48]
  ct_butterfly d0, d2, r0, r0_tw
  ct_butterfly d1, d3, r0, r0_tw
  ct_butterfly d0, d1, r1, r1_tw
  ct_butterfly d2, d3, r2, r2_tw
  vst40.u32 {d0, d1, d2, d3}, [in]
  vst41.u32 {d0, d1, d2, d3}, [in]
  vst42.u32 {d0, d1, d2, d3}, [in]
  vst43.u32 {d0, d1, d2, d3}, [in]!
  // ALTERNATIVE:
  // vstrw.u32 di, [in, #...] i=0,1,2,3
  le lr, layer45_loop
layer45_loop_end:
```

```
layer67_loop:
  vldrw.u32 data0, [in]
  vldrw.u32 data1, [in, #16]
  vldrw.u32 data2, [in, #32]
  vldrw.u32 data3, [in, #48]
  // ALTERNATIVE:
  // vld4i.u32 {d0,d1,d2,d3}, [in] i=0,1,2,3
  vldrh.u16 r0,     [r_ptr] ,#96
  vldrh.u16 r0_tw, [r_ptr, #-80]
  ct_butterfly d0, d2, r0, r0_tw
  ct_butterfly d1, d3, r0, r0_tw
  vldrh.u16 r1,     [r_ptr, #-64]
  vldrh.u16 r1_tw, [r_ptr, #-48]
  ct_butterfly d0, d1, r1, r1_tw
  vldrh.u16 r2,     [r_ptr, #-32]
  vldrh.u16 r2_tw, [r_ptr, #-16]
  ct_butterfly d2, d3, r2, r2_tw
  vstrw.u32 d0, [in], #64
  vstrw.u32 d1, [in, #-48]
  vstrw.u32 d2, [in, #-32]
  vstrw.u32 d3, [in, #-16]
  // CANONICAL:
  // vst4i.u32 {d0,d1,d2,d3}, [in] i=0,1,2,3
  le lr, layer67_loop
layer67_loop_end:
```

Listing 10: Naïve implementation of the last four layers of a Kyber NTT.

vector registers but load constants in general purpose registers, allowing them to maintain a good balance between the register files, and to merge two radix-2 layers at a time.

Coming up with the implementations from [BBMK+21, BHK+22] seems challenging due to the complex interleaving necessary to achieve good overlapping. Loc.cit. achieves this through loop unrolling and scripted register tracking. While the result is of very high performance, it is difficult both to read and adapt, and has a large code-size.

In the following, we demonstrate how SLOTHY, instantiated for the Cortex-M55 and Cortex-M85 CPUs, derives high performance Helium assembly implementations for the NTT from *readable* base implementations. Beyond being considerably less effort and easier to reproduce, the resulting code is also much smaller than that in the previous works.

6.2.2 Readable base implementations

We provide readable base implementations for the complete Kyber and Dilithium NTTs. Here, we comment on some selected aspects only.

```
.macro load_twiddles
  ldrd rt0, rt0_tw, [r_ptr],#(7*8)
  ldrd rt1, rt1_tw, [r_ptr, #(-6*8)]
  ...
  ldrd rt6, rt6_tw, [r_ptr, #(-1*8)]
.endm

.macro load_data
  vldrw.32 data0, [in]
  vldrw.32 data1, [in, #(1*16)]
  ...
  vldrw.32 data7, [in, #(7*16)]
.endm
```

```
.macro store_data
  vstrw.32 data0, [in], #128
  vstrw.32 data1, [in, #(-128+1*16)]
  ...
  vstrw.32 data7, [in, #(-128+7*16)]
.endm

layer456_loop:
  load_twiddles
  load_data
  ct_butterfly data0, data4, rt0, rt0_tw
  ct_butterfly data1, data5, rt0, rt0_tw
  ct_butterfly data2, data6, rt0, rt0_tw
  ct_butterfly data3, data7, rt0, rt0_tw
```

```
qsave QSTACK4, data4
qsave QSTACK5, data5
qsave QSTACK6, data6
ct_butterfly data0, data2, rt1, rt1_tw
ct_butterfly data1, data3, rt1, rt1_tw
ct_butterfly data0, data1, rt2, rt2_tw
ct_butterfly data2, data3, rt3, rt3_tw
qrestore data4, QSTACK4
qrestore data5, QSTACK5
qrestore data6, QSTACK6
ct_butterfly data4, data6, rt4, rt4_tw
ct_butterfly data5, data7, rt4, rt4_tw
ct_butterfly data4, data5, rt5, rt5_tw
ct_butterfly data6, data7, rt6, rt6_tw
store_data
le lr, layer456_loop
```

Listing 11: Symbolic implementation of three merged layers of a Dilithium NTT. Three stack spills are introduced to ensure realizability within the Armv8.1-M register files.

```
vldrw.u32 q0, [in_hi, #256]        //.*....
vqrdmulh.s32 q5, q0, r0_tw         //...*..
vldrw.u32 q3, [in_lo, #256]        //*.....
vmul.s32 q0, q0, r0                //..*...
// gap                             //.....*
vmla.s32 q0, q5, modulus           //....*.
layer12_loop:
  vsub.u32 q4, q3, q0          //...........*.............
  vmul.s32 q6, q4, r2          //.................*.......
  vldrw.u32 q1, [in_hi]        //..*.....................
  vmul.s32 q5, q1, r0          //....*...................
  vadd.u32 q7, q3, q0          //...........*.............
  vqrdmulh.s32 q3, q1, r0_tw   //.....*...................
  vldrw.u32 q0, [in_lo]        //*.......................
  vmla.s32 q5, q3, modulus     //......*..................
  vldrw.u32 q3, [in_lo, #272]  //.e.......................
  vmul.s32 q1, q7, r1          //............*............
  vsub.u32 q2, q0, q5          //.......*.................
  vqrdmulh.s32 q7, q7, r1_tw   //............*............
  vadd.u32 q5, q0, q5          //........*................
```

```
vmla.s32 q1, q7, modulus     //................*...........
vldrw.u32 q7, [in_hi, #272]  //...e.......................
vqrdmulh.s32 q0, q4, r2_tw   //..................*........
vadd.u32 q4, q5, q1          //.................*.........
vmla.s32 q6, q0, modulus     //...................*.......
vstrw.u32 q4, [in_lo], #16   //.....................*.....
vadd.u32 q4, q2, q6          //.....................*....
vstrw.u32 q4, [in_hi], #16   //.......................*...
vmul.s32 q0, q7, r0          //........e..................
vsub.u32 q5, q5, q1          //................*.........
vqrdmulh.s32 q1, q7, r0_tw   //.........e................
vstrw.u32 q5, [in_lo, #240]  //.........................*..
vsub.u32 q7, q2, q6          //....................*.....
vmla.s32 q0, q1, modulus     //...........e..............
vstrw.u32 q7, [in_hi, #240]  //.........................*
le lr, layer12_loop
...
```

Listing 12: Optimal (for Cortex-M55) scheduling for two layers of the NTT for CRYSTALS-Dilithium. Automatically derived via `slothy` from Listing 9.

Modular multiplication strategy. For the underlying modular multiplication, one has the choice between "rounding Montgomery multiplication", introduced in [BHK+21] and used in [BBMK+21], or "Barrett multiplication", also introduced in [BHK+21] and used in [BHK+22]. We use Barrett multiplication throughout.

Initial layers. Listing 9 shows a naïve implementation of the first two layers of the forward NTT for Dilithium, using macros to keep the implementation readable. While readable, however, it performs very poorly: The back-to-back instances of load/store and multiplication instructions mean that only little use is made of instruction overlapping.

Last layers. The last two layers pose a challenge due to permutations required to vectorize intra-vector butterflies. Since Armv8.1-M+Helium does not offer transpose-instructions like Neon, it is best to perform those permutations via the deinterleaving load/store instructions `VLD4x` and `VST4x` instead. Concretely, one can either store using `VST4x` in the penultimate loop, or load via `VLD4x` in the last loop, as indicated in Listing 10.

Merging three layers. We also study merging *three* radix-2 NTT layers a time, which for Helium has not been considered before. Merging layers is natural to save load/store operations, and particularly attractive for Kyber, which has an odd number of layers and which would otherwise require a loop for an isolated layer.

Despite the limited amount of 8 vector registers, we find that layers $4, 5, 6$ in Dilithium and $3, 4, 5$ in Kyber can be merged with only 3 stack spills per iterations — a complete load/store sequence would amount to 8 spills. Merging layers $3, 4, 5$ in Kyber allows to split its 7-layer NTT into $2 + 3 + 2$ layers. For Dilithium, we also merge layers $1, 2, 3$, but

we find that large pressure also on the *general purpose registers* requires 3 more spills. Listing 11 shows a symbolic implementation layers $3, 4, 5$ of a Dilithium NTT.

Non-canonical output format. Following [BBMK+21], we omit the inversion of the intra-vector shuffling prior to storing the result of the NTT. This allows for better comparability with [BBMK+21] and helps to reduce pressure on the register file for the last two layers. The use of this "non-canonical" ordering has no impact on polynomial multiplication in Kyber and Dilithium, as the underlying base multiplication in NTT domain is ordering-insensitive.

In case the canonical ordering is desired, one should use `VST4x` in the last two layers as shown in Listing 10. As this creates high pressure on the vector register file with the arguments to `VLD4x` and `VST4x` being constrained to $\{$`Qi,Q(i+1),Q(i+2),Q(i+3)`$\}$ for $i = 0, 1, 2, 3, 4$, it is advisable to convert the `VLD4x` in the last merge into `VST4x` in the prior-to-last one. However, in the case where the prior-to-last merge consists of three layers, this further increases the already large optimization complexity.

6.2.3 Optimized implementation

We now comment on the optimizations of our base implementations.

Performance and Heuristics. We found the optimizations of triple-merged layers with software pipelining requiring the use of heuristics, despite comparatively small numbers of instructions (the layer $1, 2, 3$ loop in Dilithium has 85 instructions, for example) when compared to some AArch64+Neon examples below which could be optimized without heuristics. We used the periodic halving heuristic in those cases (see Section 5.5).

Initial layers. Listing 12 shows the result of optimizing the first two layers of the Dilithium NTT (Listing 9) for Cortex-M55. As before, comments indicate where instructions were originally placed, and if they are early instructions for the next iteration — this time, five instructions are being pulled into the previous iteration. The body of the loop is stall-free, while the preamble and postamble have stalls, which should be attempted to be filled through further interleaving with the surrounding code. On the Cortex-M85, we also achieve a stall-free body and are left with only a single stall in the postamble.

Last layers. For the last layers, we observed that when optimizing for the Cortex-M55 microarchitecture, the use of `VST4x` in the penultimate layer-merge worked better, while for the Cortex-M85 microarchitecture, the base version using `VLD4x` in the last two layers was preferable. Listing 13 shows the last four layers of the Kyber NTT optimized for the Cortex-M55 CPU. Overall, SLOTHY is able to generate stall-free code for both Cortex-M55 and Cortex-M85 in the bodies of both loops.

Triple merged layers. We find that on the Cortex-M85 CPU, the triple merged layers admit a stall-free scheduling even *without* software pipelining, while on the Cortex-M55 CPU, only `ST-LD`-hazards remain. This leads to compact code, and the lost cycles are compensated for by loads and stores saved by merging three layers.

6.2.4 Results

Table 3 compares our optimized code to prior art. For the Dilithium NTT, we compare our $3 + 3 + 2$-layer and $2 + 2 + 2 + 2$-layer implementations to the 32-bit NTT from [BBMK+21] and to the Cortex-M4 implementation from [AHKS22]. For the Kyber NTT, we compare our $1 + 2 + 2 + 2$-layer and $2 + 3 + 2$-layer implementations to the Cortex-M4 implementations from [AHKS22, HZZ+22]; there is no prior implementation in Helium.

```
layer45_loop:
vsub.u16 q7, q1, q6          //..............*..............
vqrdmulh.s16 q4, q7, r1      //.....................*.......
vldrw.u32 q0, [r0, #32]      //.....*.....................
vmul.s16 q5, q0, r3          //.......*...................
ldrd r4, r9, [r11, #-16]     //.*.........................
vqrdmulh.s16 q2, q0, r7      //........*..................
vldrw.u32 q0, [r0]           //...*.......................
vmla.s16 q5, q2, r12         //.........*................. 
vadd.u16 q3, q1, q6          //..............*.............
vmul.s16 q7, q7, r6          //....................*......
vadd.u16 q6, q0, q5          //..........*................
vmla.s16 q7, q4, r12         //....................*......
vsub.u16 q5, q0, q5          //..........*................
vmul.s16 q2, q3, r4          //...............*...........
vadd.u16 q4, q5, q7          //.....................*.....
vqrdmulh.s16 q0, q3, r9      //...............*...........
vsub.u16 q5, q5, q7          //....................*......
vmla.s16 q2, q0, r12         //................*..........
vldrw.u32 q1, [r0, #80]      //....e......................
vsub.u16 q3, q6, q2          //..................*........
vldrw.u32 q7, [r0, #112]     //......e....................
vadd.u16 q2, q6, q2          //..................*........
ldrd r3, r7, [r11] , #24     //e..........................
ldrd r6, r1, [r11, #-8]      //..e........................
vst40.u32 {q2,q3,q4,q5},[r0] //.......................*...
vqrdmulh.s16 q0, q7, r7      //............e..............
vst41.u32 {q2,q3,q4,q5},[r0] //.......................*...
vmul.s16 q6, q7, r3          //............e..............
vst42.u32 {q2,q3,q4,q5},[r0] //.......................*...
vmla.s16 q6, q0, r12         //............e..............
vst43.u32 {q2,q3,q4,q5},[r0]! //.......................*...
le lr, layer45_loop
...
```

```
layer67_loop:
vmul.s16 q7, q0, q3          //......*....................
vadd.u16 q2, q1, q6          //..............*.............
vqrdmulh.s16 q4, q0, q4      //.......*...................
vldrw.u32 q5, [r0] , #64     //*..........................
vmla.s16 q7, q4, r12         //........*..................
vldrh.u16 q4, [r11, #-64]    //...............*...........
vmul.s16 q0, q2, q4          //.................*.........
vldrh.u16 q3, [r11, #-48]    //...............*...........
vqrdmulh.s16 q3, q2, q3      //..................*........
vadd.u16 q4, q5, q7          //..........*................
vmla.s16 q0, q3, r12         //...................*.......
vldrh.u16 q2, [r11, #-16]    //.....................*.....
vsub.u16 q3, q4, q0          //...................*.......
vstrw.u32 q3, [r0, #-48]     //.............................*..
vadd.u16 q4, q4, q0          //...................*.......
vstrw.u32 q4, [r0, #-64]     //.............................*...
vsub.u16 q4, q1, q6          //.............*..............
vqrdmulh.s16 q3, q4, q2      //....................*.......
vldrw.u32 q1, [r0, #16]      //.e..........................
vsub.u16 q5, q5, q7          //.........*.................
vldrh.u16 q6, [r11, #-32]    //.....................*.....
vmul.s16 q2, q4, q6          //....................*.......
vldrh.u16 q4, [r11, #16]     //.....e.....................
vmla.s16 q2, q3, r12         //....................*......
vldrw.u32 q6, [r0, #48]      //...e........................
vsub.u16 q7, q5, q2          //....................*......
vldrh.u16 q3, [r11] , #96    //....e.....................
vadd.u16 q5, q5, q2          //....................*....
vqrdmulh.s16 q2, q6, q4      //............e..............
vldrw.u32 q0, [r0, #32]      //..e.........................
vmul.s16 q6, q6, q3          //...........e................
vstrw.u32 q7, [r0, #-16]     //.............................*
vmla.s16 q6, q2, r12         //............e..............
vstrw.u32 q5, [r0, #-32]     //.............................*.
le lr, layer67_loop
```

Listing 13: Optimal (for Cortex-M55) scheduling for the periodic parts of the last four layers of the NTT for CRYSTALS-Kyber. Note the significant amount of interleaving.

Note that the implementations from [AHKS22, HZZ+22] store the results in canonical form, compared to [BBMK+21] and this work, which use the non-canonical order.

Our Dilithium NTTs match (within 1%) the performance from [BBMK+21] while achieving much smaller code and better maintainability and readability. We do not consider [BBMK+21] practical from the latter perspective. Compared to the Cortex-M4 implementation from [AHKS22], we note speedups of 3.97× and 4.05× for Cortex-M55 and Cortex-M85, respectively. Considering that the Dilithium NTT is a 32-bit workload theoretically amenable to a 4× SIMD speedup from Helium, this is close to optimal — and a remarkable demonstration of the power of Helium given that the Cortex-M55 CPU does only have twice the memory and ALU resources as the Cortex-M4. Finally, the performance on the Cortex-M85 CPU is slightly better yet very close to the performance on the Cortex-M55 CPU, which is expected due to their similar microarchitecture in terms of Helium (recall Section 4.2).

For the Kyber NTTs, we observe speedups of $6.36 - 6.58\times$ compared to the Cortex-M4 implementation of [AHKS22] and of $4.75 - 4.92\times$ compared to [HZZ+22], while maintaining readability and code compactness. Those gains make sense considering that the Kyber NTT is a 16-bit workload and thus amenable to 8× SIMD on Helium, and 2× SIMD on Cortex-M4 when 32-bit SIMD instructions are used. Moreover, for [HZZ+22] the speedup is smaller because the latter uses Plantard arithmetic which cannot be vectorized.

Overall, as shown in Table 3, we achieve IPCs very close to 1, despite most MVE instructions having a throughput of 1/2. The minor drop in the $1 + 2 + 2 + 2$-split for the Kyber NTT on Cortex-M55 is due to the share of loads and stores in the isolated first layer — on Cortex-M85, a stall-free scheduling exists due to load/store overlap.

6.3 Example: NTT on AArch64

In this section, we discuss the optimization of the Kyber and Dilithium NTT on AArch64, targeting the Arm Cortex-A55 and Cortex-A72 CPUs. Since this section is largely parallel to Section 6.2, we are brief and focus only on the specifics of optimizing the NTT for

		Type	Cycles	IPC	MVE eff.	Code size	R	F
32-bit Dilithium NTT				Cortex-M4				
	[AHKS22][1]	Handwritten ASM	8093	—	—	1.5 KB	✓	✗
				Cortex-M55				
	[BBMK+21][3]	Scripted ASM	2017[2]	0.97	99.4%	7.8 KB	✗	✗
	Our work 2+2+2+2 layers	slothy$_{\text{M55}}$	2073	0.96	98.9%	1.1 KB	✓	✓
	Our work 3+3+2 layers	slothy$_{\text{M55}}$	2037	0.96	99.8%	1.1 KB	✓	✓
				Cortex-M85				
	[BBMK+21][3]	Scripted ASM	1980	0.97	99.9%	7.8 KB	✗	✗
	Our work 2+2+2+2 layers	slothy$_{\text{M85}}$	2018	0.99	99.1%	1.1 KB	✓	✓
	Our work 3+3+2 layers	slothy$_{\text{M85}}$	1997	0.98	99.1%	1.1 KB	✓	✓
16-bit Kyber NTT				Cortex-M4				
	[AHKS22][1]	Handwritten ASM	5992	—	—	2.2 KB	✓	✗
	[HZZ+22]	Handwritten ASM	4474	—	—	1.7 KB	✓	✗
				Cortex-M55				
	Our work 1+2+2+2 layers	slothy$_{\text{M55}}$	1012	0.91	96.2%	0.9 KB	✓	✓
	Our work 2+3+2 layers	slothy$_{\text{M55}}$	942	0.94	99.3%	1.0 KB	✓	✓
				Cortex-M85				
	Our work 1+2+2+2 layers	slothy$_{\text{M85}}$	947	0.97	99.9%	0.9 KB	✓	✓
	Our work 2+3+2 layers	slothy$_{\text{M85}}$	910	0.96	99.4%	1.0 KB	✓	✓

[1] Result stored in canonical ordering.
[2] Measurement by ourselves.
[3] Implementation optimized for the Cortex-M55 microarchitecture.

Table 3: Comparison of Kyber and Dilithium NTTs for Armv8.1-M+Helium. R=Readable, F=Flexible μarch, MVE efficiency = `ARM_PMU_MVE_INST_RETIRED/ARM_PMU_MVE_STALL`.

AArch64. We again refer to Section 2.4 for a brief introduction to the NTT.

State of the art. [BHK+21] describes the state-of-the-art implementations of the Kyber and Dilithium NTTs for AArch64, putting emphasis on optimization for Cortex-A72. We base our implementation on their arithmetic, but create our own readable base implementation and experiment with various variations that we detail in the following.

Readable base implementation. As in Section 6.2, we experiment with different layer merges for both NTTs. Due to the large vector register file of AArch64, up to *four* initial layers can be merged without stack spills, suggesting layer splits of $4+3$ or $3+4$ for the Kyber NTT, and of $4+4$ or $3+5$ for the Dilithium NTT. Between those, the $3+4$ split for Kyber and the $3+5$ for Dilithium seem preferable since — slightly counterintuitively — they lead to smaller code size (and thus optimization complexity): The last two layers consist of intra-vector butterflies which do not require new data to be loaded and thus do not increase the vector register pressure.

We also experiment with base implementations involving *scalar* instructions to perform some loading/transposing/storing of data, the idea being to reserve the Neon units for the core arithmetic and have the scalar operations run in parallel. While scheduling such hybrid implementations by hand is very tedious and highly microarchitecture-specific, here we only need to provide the sequential implementations, which is simple — we let SLOTHY

do the interleaving work, determining if/when the hybrid approach works.

As before, we heavily use register aliases and assembly macros to make the base implementations clean and readable.

Optimized implementations. On Cortex-A55, the best implementation strategies for both Kyber and Dilithium are the scalar/Neon hybrids offloading vector *loads* to scalar instructions, but no other vector instructions. Alternatively or additionally offloading stores to scalar instructions has detrimental effect on performance. Microbenchmarks suggest that this is due to the GPR-to-Vector insert instruction `ins` dual-issuing alongside other vector instructions, while the Vector-to-GPR extraction instruction `umov` does not seem to. On the Cortex-A72 CPU, the best Kyber NTT is achieved by optimizing the basic, non-hybrid NTT implementation. For Dilithium, we achieve a slightly better performance by replacing the `st4` after the last layer by a sequence imitating its behavior using "regular" `strs` and transposition operations. Although this approach requires more instructions, it facilitates the interleaving and thus allows for fewer stalls.

Benchmarking environment. For Cortex-A55, we use a Lantronix Snapdragon 888 hardware development kit with a Qualcomm Snapdragon SM8350P SoC, featuring one Cortex-X1, three Cortex-A78, and four Cortex-A55 cores. Our benchmarking platform for Cortex-A72 is a Raspberry Pi 4. Cycle counts are obtained via PMU on Cortex-A72 and `perf` on Cortex-A55.

Results. Table 4 shows the performance of our SLOTHY-optimized NTTs in comparison with the state of the art NTTs from [BHK+21]. For Dilithium, we outperform [BHK+21] by 1.41× on Cortex-A55 and by 1.27× on Cortex-A72. For Kyber, we outperform [BHK+21] by 1.4× on Cortex-A55 and by 1.29× on Cortex-A72. Considering that [BHK+21] included microarchitecture specific optimizations for Cortex-A72, we see this as a major demonstrator of SLOTHY's capabilities.

		Type	Cycles	Code size	Readable	Flexible μarch
32-bit Dilithium NTT			Cortex-A55			
	[BHK+21]	Handwritten ASM	2436	2.3 KB	✗	✗
	Our work 3+5 layers	`slothy`$_{\texttt{A55}}$	1728	2.8 KB	✓	✓
			Cortex-A72			
	[BHK+21]	Handwritten ASM	2241	2.3 KB	✗	✗
	Our work 3+5 layers	`slothy`$_{\texttt{A72}}$	1766	2.1 KB	✓	✓
16-bit Kyber NTT			Cortex-A55			
	[BHK+21]	Handwritten ASM	1245	2.7 KB	✗	✗
	Our work 3+5 layers	`slothy`$_{\texttt{A55}}$	891	1.9 KB	✓	✓
			Cortex-A72			
	[BHK+21]	Handwritten ASM	1200	2.7 KB	✗	✗
	Our work 3+5 layers	`slothy`$_{\texttt{A72}}$	932	1.4 KB	✓	✓

Table 4: Comparison of Kyber and Dilithium NTT implementations for AArch64.

Algorithm 1 High-level description of inner loop of X25519 hybrid implementation from [Len19]. Operations executing in parallel have the executing unit in parentheses. On Neon, multiplications and squarings are vectorized across two separate operations shown on the same line.

In: In point represented by $X1$
In: Current and previous bit of the scalar: bit and $lastbit$ (processed from most to least significant)
In/Out: Temporary point $X2/Z2$; initially $X2 = 1$, $Z2 = 0$
In/Out: Temporary point $X3/Z3$; initially $X3 = X1$, $Z3 = 1$

1: $B = X2 - Z2; D = X3 - Z3$
2: $A = X2 + Z2; C = X3 + Z3$
3: **if** bit = lastbit **then**
4: $\quad F, G = C, D$
5: **else**
6: $\quad F, G = A, B$
7: $AA = F^2$ (Scalar)
8: $BB = G^2$ (Scalar)
9: $E = AA - BB$ (Scalar)
10: $Z4 = E(BB + 121666 \cdot E)$ (Scalar)
11: $DA = D \cdot A; CB = C \cdot B$ (Neon)
12: $T1 = DA + CB$ (Neon)
13: $T2 = DA - CB$ (Neon)
14: $X5 = T1^2; T3 = T2^2$ (Neon)
15: $X4 = AA \cdot BB; Z5 = X1 \cdot T3$ (Neon)
16: $Z2 = Z4; X2 = X4$
17: $Z3 = Z5; X3 = X5$

6.4 Example: X25519 on AArch64

This section discusses the use of `slothy` for the optimization of the X25519 scalar multiplication on the Cortex-A55 CPU. The complexity of this example significantly exceeds that of the previous examples, both because of code size and because of the breadth of different scalar and SIMD instructions used in it.

Introduction. X25519 (originally: Curve25519) is the name of a Diffie-Hellman key-exchange protocol proposed by Bernstein in [Ber06]. The protocol is based on the elliptic curve $E : y^2 = x^3 + 486662x^2 + x$ defined over the finite field $\mathbb{F}_{2^{255}-19}$. The most computationally expensive operation for X25519 is the variable-base-point scalar multiplication required for the calculation of the shared secret.

Target and Previous Work. Our implementation targets the Cortex-A55 in-order CPU. To the best of our knowledge, the fastest implementation of X25519 on this processor is presented in [Len19], a highly sophisticated handwritten *hybrid* implementation making use of both scalar and Neon instructions and taking into account the details of the processor's microarchitectural capabilities (we refer to [BK22, Section 4.4] for a general introduction to hybrid implementations). Originally, this implementation was developed for the Cortex-A53 processor, but as noted and leveraged by [Len19], the performance is very close to that on the Cortex-A55. [Len19] uses the popular radix $2^{25.5}$ (alternating between 25-bit and 26-bit limbs) as proposed by Bernstein [Ber06].

[Len19] is a "coarse" scalar/Neon hybrid in the sense that the top-level description of the X25519 scalar multiplication is partitioned into components some of which are implemented entirely in scalar and others entirely in Neon code; this is shown in Algorithm 1. The scalar/Neon components are then manually "zipped" together to achieve a high rate of dual-issuing. However, as a result of the high degree of optimization and especially the manual interleaving, the code is not intuitively readable.

Since [Len19] already achieves a high IPC of 1.92 in the main loop (the 958 instructions take 498 cycles), our goal here is to merely *match* its performance, but on the basis of a *readable*, non-interleaved base implementation, with `slothy` taking care of the interleaving. This simplifies code audit as well as the transfer to other microarchitectures.

Readable Base Implementation. We focus on the main loop of the scalar multiplication as it vastly dominates the performance. In a nutshell, to obtain our base implementation, we take the implementation from [Len19] and transform it into a simplified and readable base implementation by using macros and by de-interleaving the instructions. The only semantic change we apply is aligning `X1`, `A`, and `B` stored on the stack to 16-byte boundaries. We apply this change as our experiments suggest that `stp xA, xB, [sp, #<imm>]` has a longer latency when `imm` is not a multiple of 16. Besides that, we do not semantically

```
.macro scalar_mul sC, sA, sB
  scalar_mul_inner                                \
    \sC\()0, \sC\()1, \sC\()2, ... \sC\()9,       \
    \sA\()0, \sA\()1, \sA\()2, ... \sA\()9,       \
    \sB\()0, \sB\()1, \sB\()2, ... \sB\()9
.endm

.macro scalar_mul_inner                           \
  sC0, sC1, sC2, ..., sC9,                        \
  sA0, sA1, sA2, ..., sA9,                        \
  sB0, sB1, sB2, ..., sB9

  umulll  X<tmp_9>, W<\sA1>, W<\sB8>
  umaddl X<tmp_9>, W<\sA2>, W<\sB7>, X<tmp_9>
  ...
  umaddl X<tmp_9>, W<\sA9>, W<\sB0>, X<tmp_9>
  umaddl X<tmp_9>, W<\sA0>, W<\sB9>, X<tmp_9>

  mul     W<tmp_tw_9>, W<\sA9>, W<const19>

  umulll  X<tmp_8>, W<\sA1>, W<\sB7>
  umaddl X<tmp_8>, W<\sA3>, W<\sB5>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA5>, W<\sB3>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA7>, W<\sB1>, X<tmp_8>
  umaddl X<tmp_8>, W<tmp_tw_9>, W<\sB9>, X<tmp_8>
  add     X<tmp_8>, X<tmp_8>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA0>, W<\sB8>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA2>, W<\sB6>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA4>, W<\sB4>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA6>, W<\sB2>, X<tmp_8>
  umaddl X<tmp_8>, W<\sA8>, W<\sB0>, X<tmp_8>
  ...
.endm
```

```
// ...
mainloop:
  vector_sub vB, vX2, vZ2
  vector_sub vD, vX3, vZ3
  vector_add vA, vX2, vZ2
  vector_add vC, vX3, vZ3
  vector_cmov vF, vA, vC
  vector_to_scalar sF, vF
  vector_cmov vG, vB, vD
  vector_to_scalar sG, vG
  vector_transpose vAB, vA, vB     // (B|A)
  vector_transpose vDC, vD, vC     // (C|D)
  scalar_sqr sAA, sF
  scalar_sqr sBB, sG
  // store sAA as A; store sBB as B
  scalar_sub sE, sAA, sBB
  scalar_addm sBB, sBB, sE, 121666
  scalar_mul sZ4, sBB, sE
  vector_mul vADBC, vAB, vDC
  // T1 = DA + CB; T2 = DA - CB; Repack T=(T1|T2)
  vector_addsub_repack vT, vADBC
  vector_sqr vTA, vT
  // copy lower of vTA (T1^2) to vX3
  vector_load_lane vTA, STACK_A, 1  // load A in vT[1]
  vector_load_lane vBX, STACK_B, 1  // load B in vBX[1]
  vector_load_lane vBX, STACK_X, 0  // load X1 in vBX[0]
  vector_mul vX4Z5, vTA, vBX
  scalar_to_vector vZ2, sZ4
  // copy lower of vX4Z5 to vX2
  // copy upper of vX4Z5 to vZ3
  // scalar bit and counter logic
  bpl   mainloop
// ...
```

Listing 14: Excerpts of our clean X25519 scalar multiplication "base" implementation, obtained from [Len19] via extensive use of macros, aliases, and complete de-interleaving.

change the implementation — that is, we use exactly the same instructions and only alter their order and (sometimes) their use of registers.

We introduce macros for performing base field multiplication and squaring using either scalar instructions (`scalar_mul`, `scalar_sqr`) or Neon instructions (`vector_mul`, `vector_sqr`). Additionally, we introduce macros for subtraction (`scalar_sub`) and addition with multiplication by a constant (`scalar_addm`) using scalar instructions. Each of these macros operates on a large number of registers, but we use register "group aliases" prefixed by `s` (for scalar register groups) or `v` (for vector register groups) to avoid long argument lists: For example, if `sA0-sA9` refers to a set of scalar registers representing a 255-bit scalar, we pass `sA` to our macros and internally expand to `sA0-sA9` (by writing `sA\()0` etc.) — see Listing 14 (left). Using these macros, we can then implement the pseudocode from Algorithm 1 relatively straight-forwardly as shown in Listing 14 (right). We separate operations performed on the scalar and vector units, resulting in much more readable and maintainable code. However, due to the strict separation of scalar and vector code, the base implementation performs poorly esp. on in-order CPUs like the Cortex-A55.

6.4.1 Auto-optimized implementation

When using `slothy` to optimize our X25519 base implementation, one faces multiple challenges: First, with the inner loop consisting of 958 instructions (much larger than the 225 instructions of our prior largest example), optimizing it all at once via the vanilla SLOTHY approach results in a model too complex to be feasible for CP-SAT. Second, the code uses a large number of scalar, 64-bit Neon and 128-bit Neon instructions, and thus requires a much more complete model of the architecture and microarchitecture.

Heuristics. We use a variety of heuristics introduced in Section 5.5 to keep the SLOTHY optimization steps feasible. First, we apply the CFG-based preprocessing to achieve a coarse interleaving based on depths. We then run multiple optimization passes using the splitting heuristic, using `move_stalls_to_top` and `move_stalls_to_bottom` to "comb" stalls towards the middle of the code where they can be absorbed. During

this phase, we ignore latencies as we are only interested in achieving a good utilization of the execution units. Finally, we take a last pass over the whole code using the splitting heuristic, this time taking latencies into account.

Temporary registers. Throughout the computation, various temporary registers are required in both the scalar and vector code. While the current implementation already has a valid assignment of architectural registers, we can leave this register selection to `slothy` allowing more flexibility as explained in Section 5.3. As the splitting heuristic is not able to work with symbolic registers, we let `slothy` preprocess the entire loop body to replace all symbolic registers (by setting `constraints.allow_reordering=False` and `constraints.functional_only=True`).

AArch64 Register views. Another challenge when optimizing AArch64 code with SLOTHY is that AArch64 allows access to parts of registers: For example, the lower 32-bit of the 64-bit register `x0` can be accessed using the `w0` alias, and similarly `d0` corresponds to the lower 64-bit of the 128-bit vector register `v0`. The core of SLOTHY is not aware of these register views and only models the full registers. To accommodate, we extend the instruction parser to replace those register views with the corresponding full register names before passing the instructions to SLOTHY, and re-apply the views when writing out the result. For symbolic registers, we extend the parser to allow accessing different views of the symbolic register: For example, to access the `w`-view of a symbolic general purpose register with name `reg`, one can write `W<reg>`.

6.4.2 Results

Benchmarking environment. For the Cortex-A55 processor, we use the same platform as in Section 6.3 (a Qualcomm Snapdragon SM8350P). To verify that our changes do not lower the performance on the target platform of [Len19], we also benchmark on a Raspberry Pi 3 Model A+ which comes with four 1.4 GHz Cortex-A53 cores.

Performance. Table 5 contains the cycle counts for a single variable-base-point scalar multiplication on the Cortex-A53 and Cortex-A55 CPUs. Note that we do not apply optimizations specific to Cortex-A53, i.e., we benchmark the Cortex-A55-optimized code on the Cortex-A53. Our implementation is slightly faster than the implementation from [Len19], while maintaining exactly the same instructions and code size. Around 1 000 cycles of the improvement are due to aligning the stack, while the rest is due to `slothy`. Our main loop takes 483 cycles on the Cortex-A55, which is 15 cycles less than the main loop of [Len19], and thereby achieves a near-perfect 1.98 IPC.

Optimality. As noted in Section 5, global optimality of SLOTHY-optimized source code is no longer guaranteed once heuristics are used. Yet, the near-perfect IPCs obtained in this example demonstrate that SLOTHY remains a very powerful tool even for large kernels requiring extensive use of heuristics.

7 Conclusion

In this work, we studied a workflow for the development of high performance assembly based on constraint programming: Beyond achieving highest performance through micro-architectural optimizations, our flow keeps the code artifact clean and maintainable, and produces results that can be audited without having to trust the optimizer.

At the core, our workflow is based on a (micro)architecture agnostic approach SLOTHY to modelling the simultaneous optimization of instruction scheduling, register allocation

		Type	Cycles	Code size	Readable	Flexible μarch
X25519			Cortex-A55			
	[Len19]	Handwritten ASM	143 849	5.8 KB	✗	✗
	Our work	`slothy`$_{\texttt{A55}}$	139 752	5.8 KB	✓	✓
			Cortex-A53			
	[Len19]	Handwritten ASM	144 168	5.8 KB	✗	✗
	Our work	`slothy`$_{\texttt{A55}}$	140 096	5.8 KB	✓	✓

Table 5: Comparison of implementations of the variable-base-point scalar multiplication of X25519 for AArch64 . We measure our code optimized for the Cortex-A55 also on the Cortex-A53 that was used by [Len19].

and software pipelining as a constraint satisfaction problem. We provide an implementation of SLOTHY and instantiate it with Armv8.1-M+Helium + Cortex-M55 + Cortex-M85 as well as AArch64+Neon + Cortex-A55 + Cortex-A72. We showcase the power of our approach by optimizing three real-world workloads from Digital Signal Processing and cryptography, matching or improving upon prior art in all cases.

We believe that our workflow is a promising avenue to overcome the trade-off between fast and clean implementations, and encourage further research into the application of SLOTHY to other (micro)architectures and workloads, as well as the development of mechanized formal verification of the optimizations.

Acknowledgments

Matthias J. Kannwischer was supported by the Taiwan Ministry of Science and Technology through grant 109-2221-E-001-009-MY3, Academia Sinica Investigator Award AS-IA-109-M01, and the Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

We thank Joseph Yiu for helping with various questions around the Cortex-M55 and Cortex-M85 CPUs as well as the setup of our MPS3-based benchmarking platform.

References

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. Faster kyber and dilithium on the cortex-m4. In *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, page 853–871, Berlin, Heidelberg, 2022. Springer-Verlag. 32, 33, 34

[Arma] Arm Limited. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest. 6

[Armb] Arm Limited. Arm Cortex-A Series Programmer's Guide for Armv8-a. https://developer.arm.com/documentation/den0024/a/. 7, 8

[Armc] Arm Limited. Arm Cortex-M4 Processor Technical Reference Manual Revision r0p1. https://developer.arm.com/documentation/100166/0001. 20

[Armd] Arm Limited. Arm Cortex-M7 Devices Generic User Guide r1p2. https://developer.arm.com/documentation/dui0646/latest. 20

[Arme] Arm Limited. ARM-Software Repository. https://github.com/ARM-Softwareegithub.com/ARM-Software/EndpointAI. 14, 28, 29

[Armf] Arm Limited. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/latest. 6, 7

[Armg] Arm Limited. Blog series: "making Helium: Why not just add Neon?". Part 1: https://community.arm.com/developer/research/b/articles/posts/making-helium-why-not-just-add-neon, Part 2: https://community.arm.com/developer/research/b/articles/posts/making-helium-sudoku-registers-and-rabbits, Part 3: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles, Part 4: https://community.arm.com/developer/research/b/articles/posts/making-helium-going-around-in-circles. 7

[Armh] Arm Limited. CMSIS-DSP Repository. https://github.com/ARM-software/CMSIS-DSP. 28, 29

[Armi] Arm Limited. Cortex-A55 Software Optimization Guide. 19

[Armj] Arm Limited. Cortex-A72 Software Optimization Guide. 20

[Armk] Arm Limited. Cortex-M55 Software Optimization Guide. 19

[Arml] Arm Limited. Cortex-X1 Software Optimization Guide. 21

[Armm] Arm Limited. F32 floating-point FFT implemnentation in handwritten assembly, for Cortex-M55. https://github.com/ARM-software/EndpointAI/blob/aac5349a96c1ac8212fa7934883142d4ec66fa84/Kernels/ARM-Optimized/DSP/Source/TransformFunctions/arm_cfft_utils.h#L212. 29

[Armn] Arm Limited. F32 floating-point FFT implemnentation in handwritten assembly, for Cortex-M85. https://github.com/ARM-software/EndpointAI/blob/aac5349a96c1ac8212fa7934883142d4ec66fa84/Kernels/ARM-Optimized/DSP/Source/TransformFunctions/arm_cfft_utils.h#L89. 29

[Armo] Arm Limited. F32 floating-point FFT implemnentation in intrinsics. https://github.com/ARM-software/CMSIS-DSP/blob/633b5284212e4e5dd5adb1e97039d1a8185a7eaa/Source/TransformFunctions/arm_cfft_radix4_f32.c#L111. 29

[Armp] Arm Limited. Neon Programmer Guide for Armv8-A Coding for Neon. https://developer.arm.com/documentation/102159/0400. 8

[Armq] Arm Limited. Q.31 fixed-point FFT implemnentation in handwritten assembly, for Cortex-M55 and Cortex-M85. https://github.com/ARM-software/EndpointAI/blob/aac5349a96c1ac8212fa7934883142d4ec66fa84/Kernels/ARM-Optimized/DSP/Source/TransformFunctions/arm_cfft_utils.h#L374. 29

[Armr] Arm Limited. Q.31 fixed-point FFT implemnentation in intrinsics. https://github.com/ARM-software/CMSIS-DSP/blob/633b5284212e4e5dd5adb1e97039d1a8185a7eaa/Source/TransformFunctions/arm_cfft_q31.c#L38. 29

[Arms] Arm Limited. Whitepaper: Introducing Cortex-M55. https://www.arm.com/resources/white-paper/cortex-m55-introduction. 7

[Armt] Arm Limited. Whitepaper: Introduction to the Armv8.1-M Architecture. https://www.arm.com/resources/white-paper/intro-armv8-1-m-architecture. 7

[BBMK+21] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):482–505, Nov. 2021. 7, 29, 30, 31, 32, 33, 34

[Ber] Daniel J. Bernstein. qhasm: tools to help write high-speed software. http://cr.yp.to/qhasm.html. Accessed 2023-01-14. 2

[Ber06] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*, pages 207–228. Springer, 2006. 36

[BHK+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, Nov. 2021. 5, 29, 31, 34, 35

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Lorenz Panny, and Bo-Yin Yang. Efficient multiplication of somewhat small integers using number-theoretic transforms. In *Advances in Information and Computer Security: 17th International Workshop on Security, IWSEC 2022, Tokyo, Japan, August 31 – September 2, 2022, Proceedings*. Springer-Verlag, 2022. 29, 30, 31

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In *Progress in Cryptology – INDOCRYPT 2022*, 2022. 36

[Emb15] Embecosm Limited, supported by Innovate UK. Superoptimization – Feasibility Study, 2015. https://www.embecosm.com/appnotes/ean15/ean15.html. 9

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996. 4, 9

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, Aug. 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833. 32, 33, 34

[KL99] Daniel Kästner and Marc Langenbach. Code optimization by integer linear programming. pages 122–136, 03 1999. 4, 9

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4. 20

[KW98] Daniel Kästner and Reinhard Wilhelm. Operations research methods in compiler backends. *Mathematical Communications (mc@mathos.hr); Vol.3 No.2*, 01 1998. 4

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 318–328, New York, NY, USA, 1988. Association for Computing Machinery. 5

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, 2022. https://pq-crystals.org/dilithium. 10

[Len19] Emil Lenngren. AArch64 optimized implementation for X25519, 2019. https://github.com/Emill/X25519-AArch64/blob/master/X25519_AArch64.pdf. 19, 20, 36, 37, 38, 39

[LGAV96] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, pages 80–86, 1996. 5

[Mas87] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126. Association for Computing Machinery, 1987. 4, 8, 9

[McK65] William M McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965. 11

[ML93] Bart MacCarthy and Jiyin Liu. Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. *International Journal of Production Research - INT J PROD RES*, 31:59–79, 01 1993. 8

[NG07] Santosh G. Nagarakatte and R. Govindarajan. Register allocation and optimal spill code scheduling in software pipelined loops using 0-1 integer linear programming formulation. pages 126–140. Springer Berlin Heidelberg, 2007. 4

[NIS16] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. 2, 9, 10

[PF] Laurent Perron and Vincent Furnon. Google OR-Tools. https://developers.google.com/optimization/. 9

[PFK+22] Thomas Prest, Pierre-Alain Fouque, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, 2022. https://falcon-sign.info. 9

[Pin16] Michael L. Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Springer, Berlin, Heidelberg, 5 edition, 2016. 8

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, page 63–74, New York, NY, USA, 1994. Association for Computing Machinery. 5

[RG81] B. Rau and Christopher Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. volume 12, 12 1981. 5

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, 2022. https://pq-crystals.org/kyber. 10

[SCC+17] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017. 4, 9, 11

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery. 4, 9

[TA16] Andrew S. Tanenbaum and Todd Austin. *Structured computer organization.* Pearson India, 6th ed edition, 2016. 6

[WGB94] T.C. Wilson, G.W. Grewal, and D.K. Banerji. An ilp solution for simultaneous scheduling, allocation, and binding in multiple block synthesis. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 581–586, 1994. 4, 9