Applied Compiler Optimizations for Proving Code

by

Ricardo Ruiz

S.B., Massachusetts Institute of Technology (2024)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by: Ricardo Ruiz
Department of Electrical Engineering and Computer Science
August 8, 2025

Certified by: Saman Amarasinghe
Professor of Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Applied Compiler Optimizations for Proving Code

by

Ricardo Ruiz

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The recent popularity of massively distributed, trustless systems has created a demand for cryptographic proofs: systems to prove that a piece of data is a valid output for a given program. These systems exist, but face very high runtimes for the generation of proofs. Significant effort has been invested in optimizing the prover systems, but relatively less has been focused on optimizing the code that gets read as an input. This paper proposes a new approach to optimizing prover systems by modifying the compiler to produce proof-ready code. It proposes a benchmarking framework for comparing the relative proof costs of RISC-V instructions; the resulting analyis find that shift instructions do not offer heavy savings over multiplication. The finding suggests that strength reduction, a fundamental optimization in modern compilers, can sabotage end-to-end performance. The paper proposes methods for applying this knowledge to better optimize code, leaving the door open for future researchers to continue to make code proofs more performant and accessible.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

Chapter 1

Introduction

The recent growth of massively distributed systems has enabled the development of increasingly complex user-facing applications. Given the low trust model inherent to these systems, the burden falls on to the application code to prove the results of their computation. Frameworks for building these proofs exist: Succint Labs' SP1 is one of them. The SP1 toolchain [1] allows developers to write and generate proofs for arbitrary Rust code. With it, developers can innovate on novel applications in the growing Web3 space.

Code proof performance places a significant constraint on this story. Binaries that hundreds of microseconds to run on a developer's machine can take minutes or even hours to prove locally. There is work being done to mitigate this: SP1 remains under active development, and Succint Labs has built out a prover network to share costs among a pool of machines. This understanding of code as a black box, seeks to modify the prover architecture to improve proof performance on a fixed binary. This paper explores the opposite approach: treating the prover as a black box and modifying the incoming code instead.

The key to this approach lies in the Rust compiler. Modern compilers are very good at optimizing code for runtime performance, eliminating unnecessary cycles wherever possible. In specific contexts, these optimizations can be an example of overfitting and detract from the developers goals. A common example is the GPU context: GPUs have fundamentally different

strengths and weaknesses compared to a CPU, to the point where GPU-specific compilers have become their own field. This paper suggests that this overfitting problem is happening in the context of code proofs. Certain compiler optimizations (namely strength reduction) done with the goal of saving microseconds at runtime may end up costing the develop hours of prove time. This paper documents these findings to facilitate saving developer time from over-compiled code.

My specific contributions are:

- Discovering that the strength reduction of constant integer multiplication can have significant negative effect on prove time.
- Proposing tools and solutions to fix the problem of overcompilation, specifically of strength reduction.
- Finding a relative proof cost for instructions in the RISC-V ISA to inform which instructions to use and to avoid.
- Investigating the efficacy of explicit vs. implicit memory loads to determine if staged compilation is worthwhile.

The rest of this thesis is organized as follows:

Chapter 2 - SP1 Infrastructure gives a brief explanation of code proofs being generated and provides a walkthrough of all the relevant SP1 system components.

Chapter 3 - Experimental Design describes the high-level approach for benchmarking the individual instructions of the RISC-V ISA.

Chapter 4 - Strength Reduction describes the main positive result of the paper: strength reduction is suboptimal for proof performance. The section also describes attempts at applying this finding.

Chapter 5 - Instruction Benchmarking proposes a relative prove cost table for the arithmetic instructions of the RISC-V ISA. The chapter describes a major bug which pollutes

these findings, efforts to account for the bug in the experimental results, and whether these results agree with Chapter 3.

Chapter 6 - Register Loading investigates how register loads impact prove performance. It explores the difference between implicit and explicit loading, but the experimental data for this section is ultimately inconclusive.

Chapter 7 - Conclusions andn Future Work concludes with potential avenues for future work on this topic.

Chapter 2

SP1 Infrastructure

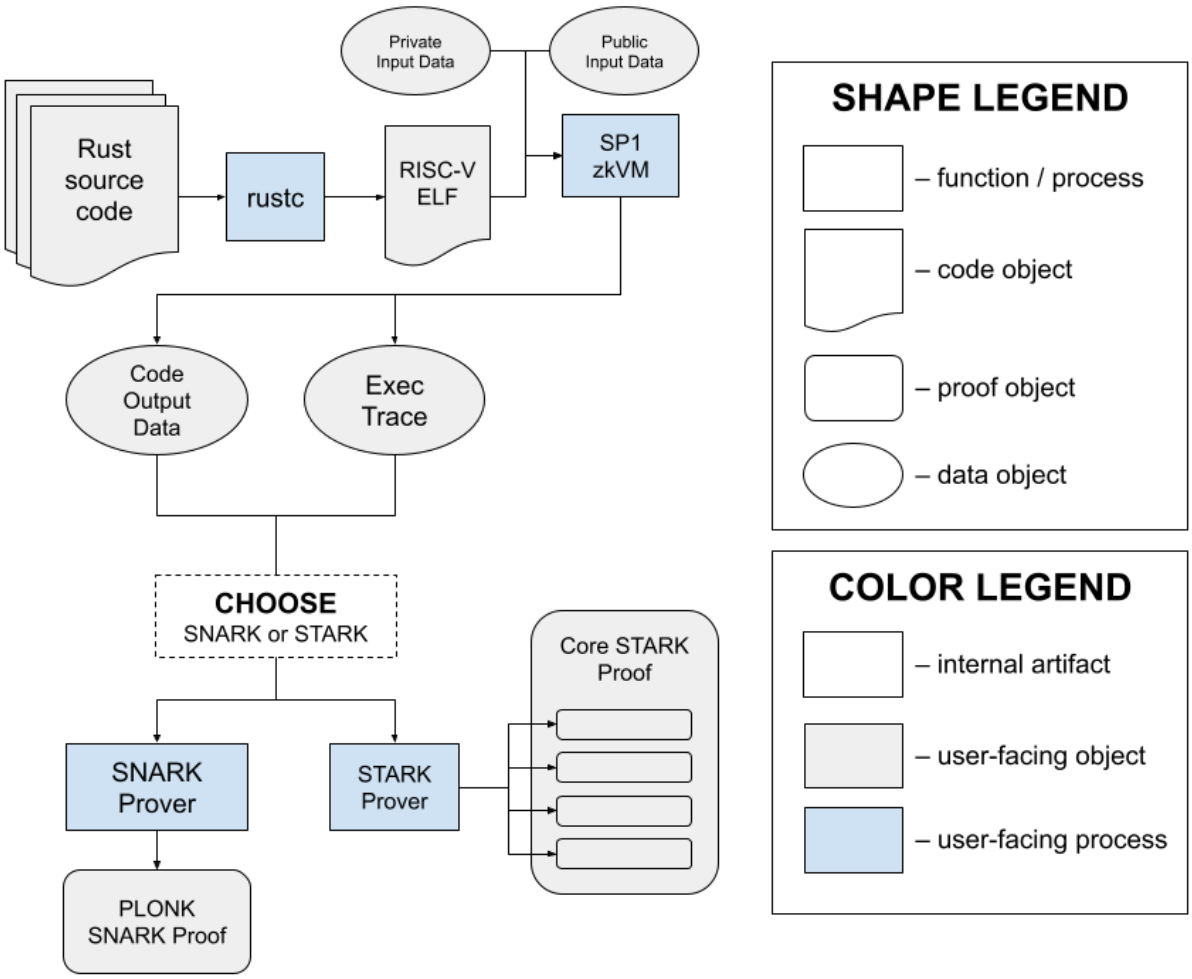


Figure 2.1: Map of SP1 prover infrastructure

This chapter provides an end-to-end analysis of components of the SP1 prover system. It starts with a brief explanation of the Rust compiler, continues into the execution virtual machine, and then explores the different proofs that can be generated from program traces. The diagram in Fig. 2.1 gives a visual overview of this process from start to finish. Each of the following sections covers a component of this prover system and includes a more detailed dataflow diagram. These diagrams use the same legend as Fig. 2.1 and show time flowing from left to right.

Unless explicitly stated otherwise, all zkVM executions and proof generations are run on a single thread of a 2.4 GHz Intel Xeon E5-2695 v2 machine with an allowance of 96 GB of main memory. All physical hardware benchmarks run on the same machine, with the RISC-V instructions translated into x86 equivalents.

2.1 Compiler Toolchain

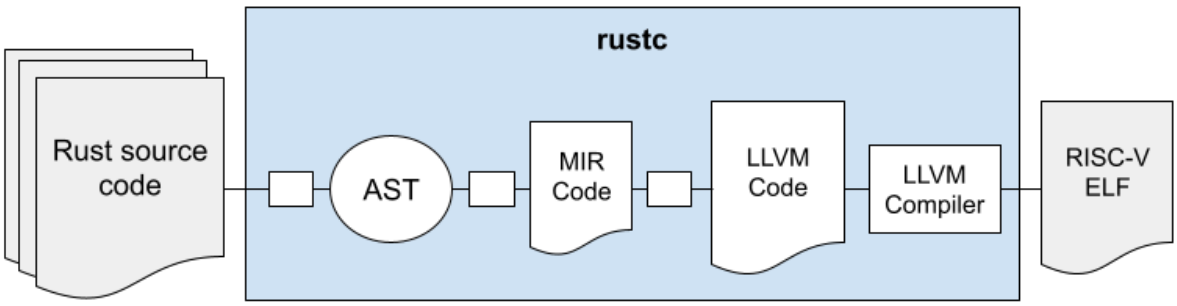


Figure 2.2: Dataflow visualization for SP1 compiler toolchain

The SP1 prover toolchain (Fig. 2.2) compiles Rust code into a statically linked RISC-V executable and linkable (ELF) file. This includes the program and all its dependencies in a single file, which can be read by the zkVM (Section 2.2).

SP1 CLI provides the `cargo prove build` command to compile programs into the RISC-V ELF format. This command invokes a custom build of the `rustc` compiler using the

`riscv32-im-succint-zkvm-elf` compilation target. The Rust compiler works by lowering Rust code into MIR, a mid-level representation which simplifies many of the language's high-level constructs. This MIR code is then transformed into LLVM, at which point the LLVM compiler infrastructure is used to generate the final assembly.

The command itself does not specify an optimization level to the compiler and only requires the `lower-atomic` LLVM pass. This LLVM pass replaces all atomic intrinsic instructions with their non-atomic equivalents. Since the zkVM does not virtualize a multi-threaded execution environment, the atomic instructions would only add unnecesssary synchonization overhead.

2.2 The SP1 zkVM

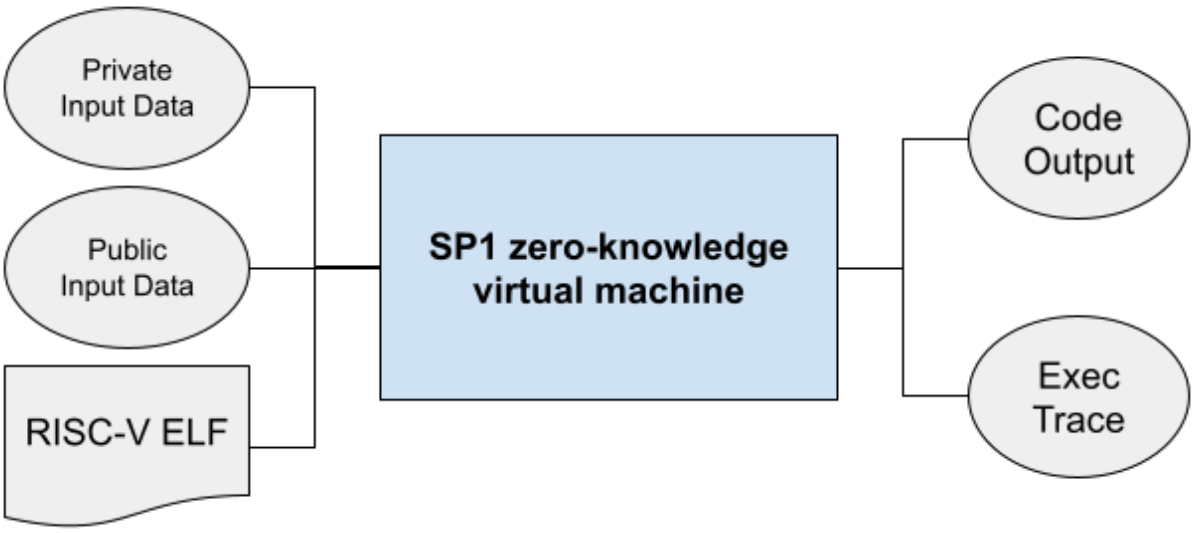


Figure 2.3: Dataflow visualization for the SP1 zkVM

SP1 provides a zero-knowledge virtual machine (zkVM) for the execution of binaries. The dataflow diagram for the zkVM is shown in Fig. 2.3. In a standard execution environment, the machine is free to discard the internal state of the computation as the program advances. This is not the case in the context of proof generation. Proof generation requires an underlying

execution trace to log the internal state of the computer thoguhout code execution. The zkVM tracks this internal state by keeping track of tables corresponding to instruction execution, control flow branches, memory accesses, etc. These tables are critical for the translation of RISC-V code into arithmetic circuits (AIR) as detailed in the next section.

The use of a zkVM instead of a physical execution machine has several implications for the performance of binaries. The first of these concerns the memory system. In hardware, memory fetches have to navigate the memory hierachy to fetch data from disk. The latency of any particular fetch depends is variable and depends on both the machine's cache and the program's access patterns. The zkVM virtualizes the memory system, creating much more constant patterns for fetch latency. This is explored further in Chapter 6. At the individual instruction level, the zkVM might incur different latencies for instructions compared to hardware. Exploting gaps in these patterns can provide opportunities for optimization.

2.3 SP1 Proofs

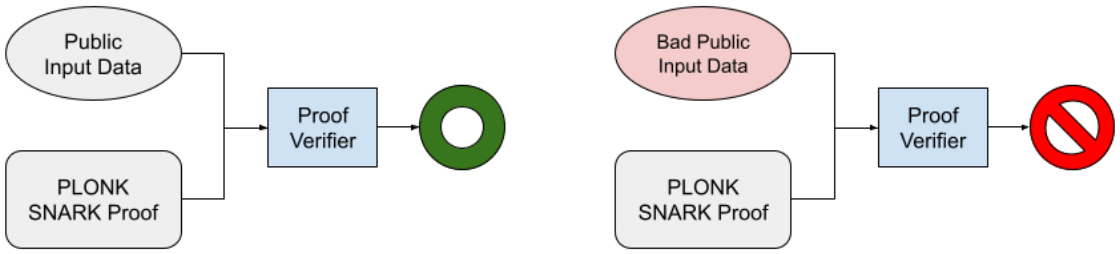


Figure 2.4: Zero-knowledge proof verification system

The SP1 prover toolchain is designed to produce non-interactive zero-knowledge proofs. These proofs assume two parties: the prover and the verifier. The prover possesses some statement which the verifier wants to check, and the proof allows the verifier to do so. SP1 proofs are characterized as:

- **Non-Interactive**: the prover and verifier only need to communicate once

	zk-STARK	**zk-SNARK**
Proof Size	Scales with input	Constant
Trusted Setup	Not required	Required

Figure 2.5: Differences between zk-STARK and zk-SNARK proofs

- **Zero-Knowledge**: the verifier only learns the statement is valid

In the context of programs, the statements being proved are the executions of compiled binaries. The prover party runs their computation using a combination of public and private values to generate a code proof (Fig 2.1). The verifier can then use this proof and the public input values (Fig 2.4) to verify the computation, without ever learning the private input data. In practice, this verification step is orders of magnitude faster than the execution of the original binary.

Zero-knowledge non-interactive proofs are particularly attractive for blockchain applications that run in low-trust distributed environments. In lieu of a central trusted authority, code proofs allow independent actors to do trusted computations. Non-interactive proofs reduce the overhead required for communication between two parties, and zero-knowledge proofs allow the prover to hide information from the verifier. This allows the proof of computations involving secret data, such as passwords or wallet addresses. Systems like ZCash [2] use this property to put transaction verifications on a blockchain while protecting sensitive transaction details.

We are concerned with two zero-knowledge proof classes produced by SP1: Scalable Transparent Argument of Knowledge (zk-STARK) and Succint Non-interactive Argument of Knowledge (zk-SNARK). The table in Fig 2.5 summarizes the differences between the proofs.

The large size of zk-STARK proofs makes them less practical for real-world applications than zk-SNARKs; the former is often measured in tens or hundreds kB while the latter is measured in tens or hundreds of bytes [3]. The SP1 toolchain generates a zk-STARK proof in the process of generating zk-SNARK proof, so zk-SNARK prove times will be longer. This chapter explores the generation of both.

2.4 zk-STARK Proof Generation

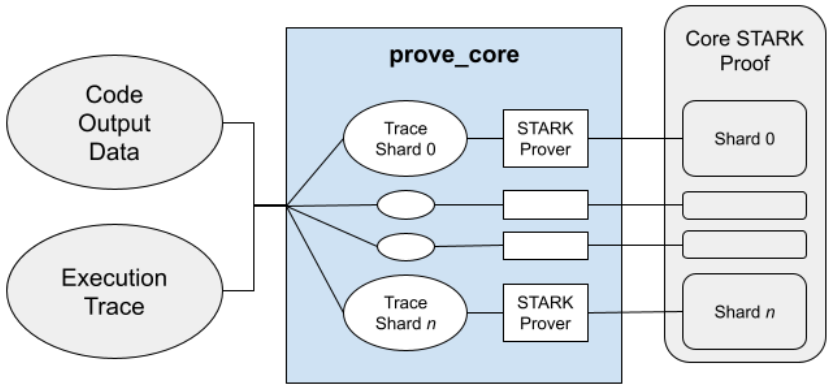


Figure 2.6: Dataflow visualization for the zk-STARK proof generator

The generation of Core zk-STARK proofs (Fig. 2.6) starts with a trace of the program execution. This trace is split into shards, with a separate zk-STARK proof generated for each shard. Within the shard, the RISC-V assembly instructions are translated into an arithmetic intermediate representation (AIR) which encodes the code logic into an arithmetic relationship between data. The data points used by an AIR instruction are **nodes**, and the AIR describes how to relate the nodes to each other.

Each broad class of AIR instruction corresponds an SP1 **prover chip**. For example, the `add`, `addi`, `sub`, and `subi` instructions are all represented by the `AddSub` prover chip. These prover chips are virtual circuits used to encode the relationship between nodes in a way the

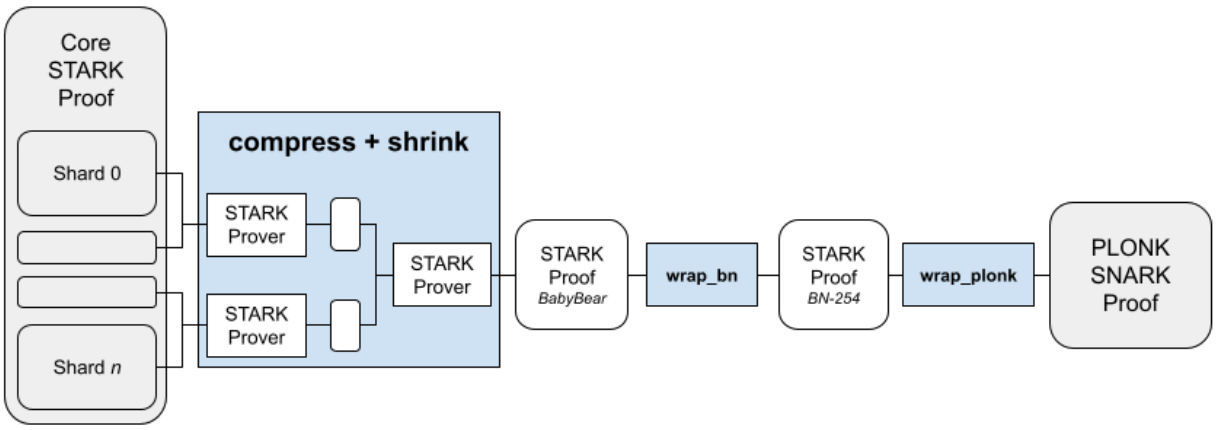


Figure 2.7: Dataflow visualization for zk-SNARK proof generation

prover can understand. The total area used in the prover circuit greatly determines the cost of the proof generation, so keeping area down is desirable. The area of a given prover chip is a product of two variables: width and height.

The width of a prover chip is independent of the execution trace and generally corresponds to the complexity of the underlying AIR instruction. More complex instructions – such as divisions and control flow branches – have wider chips.

The height of a prover chip depends on the execution trace. Each prover chip can be thought of as a matrix with width described above. New invocations of AIR instructions correspond to rows in this matrix, so larger binaries will have taller prover chips.

The task of optimizing the proof generation time is done by minimizing the total area of the prover circuit. Operations like strength reductions, loop unrolling, and hoisting affect the way these circuits are built, but the compiler's cost model may not be consistent with the prover chip dimensions. For example: the compiler may substitute one wide instruction with several narrow instructions. If the total area of the narrow instructions adds up to more than the area of the single wide instruction, prove time will suffer.

2.5 zk-SNARK Proof Generation

Given the sharding process described in the previous section, the Core zk-STARK proofs are often impractical for scalable deployment. To address this issue, SP1 provides a process to turn generate zk-SNARK proofs using the Plonky3 [4] toolchain, which uses the PLONK proof construction to optimize for low prover overheads [5]. This process has five named phases, visualized in Fig. 2.7:

- `prove_core`: generation of sharded zk-STARK proof (Fig. 2.6)
- `compress`: compression into single proof
- `shrink`: cleanup on large zk-STARK proof
- `wrap_bn254`: change underlying proof field
- `wrap_plonk_bn254`: PLONK proof generation

The `prove_core` phase is simply the process described in Section 2.4, generating a sharded zk-STARK proof. The `compress` phase uses the prover to combine the shards into a single, large zk-STARK proof. This is done by recursively invoking the prover over the proof shards. These types of recursive proof protocols allow for incremental verification of large binaries, improving proof efficiency [6]. The resulting zk-STARK proof is massaged a bit with the `shrink` function. The time cost of this process is dominated by the number of shards in the execution trace: `compress` time is linearly related with shard number with `shrink` inducing a small constant-time cost at the end.

The zk-STARK proof generated from the `prove_core` process uses the BabyBear prime finite field. This field is relatively small, using only 31 bits to encode possible values. The small size of the field and choice of constitutent primes facilitate the Reed-Solomon proofs used by SP1 to maximize proving efficiency. Once the proof has been wrapped into the larger field, the PLONK prover can generate a zk-SNARK proof.

Chapter 3

Experimental Design

This chapter briefly covers the ways that the SP1 architecture can be leveraged to optimize code at compile-time for proof performance.

The cost of proving a binary is often orders of magnitude more expensive then execution within the zkVM. Given that zkVM execution incurs additional overhead on top of code execution for witness generation (Section 2.2), the proof cost for a binary will dominate the execution cost on physical hardware. Generating a zk-SNARK proof on top of a zk-STARK proof further increases end-to-end times. As an example, the table in Fig. 3.1 shows how dramatically the costs for execution and proving compare for a sample binary from Chapter 4. All times are in seconds.

Size	x86 Exec Time	zkVM Exec Time	zk-STARK Time	zk-SNARK Time
Small	0.000358	0.218	2158	9323
Medium	0.000601	0.293	2156	9340
Large	0.000881	0.415	4708	12488

Figure 3.1: End-to-end costs for a three binaries

One way to reduce the proof cost of code is simply to use less of it. Smaller binaries induce smaller execution traces and require less overhead. During execution (a prerequisite to any proof), the zkVM must track the internal state of the machine. Larger binaries (and by extent, traces) tend to have more variables and internal state to keep track of. This methods

is particularly important for generating small, user-friendly PLONK proofs. These proofs use recursive proving to combine zk-STARK proofs into a single output: larger execution traces and more shards directly correpond to greater prove times for this phase.

The other way to reduce the proof cost is to minimizes the total area footprint of the prover circuit. Using smaller traces already takes a step in this direction: less assembly instructions translate to less AIR instructions, which occupy less prover chip height. Prover area can also be optimized by cleverly balancing the width of prover chips. If a single, wide chip can be replaced by many narrow chips (or vice-versa) such that the total circuit area decreases, the prove time should fall. This is especially true for Core zk-STARK proofs, where less external variables contribute to prove times.

Optimizing compilers already make both these changes to minimize the run time of a program. Strength reduction is a key example of this technique, replacing one expensive instruction (multiplication) with more, cheaper instructions (shifts and adds). These optimization are tuned for execution time in hardware instead of the much larger proving time. This chapter aims to establish a relative cost model in the context of proof generation to inform a properly tuned suite of compiler optimizations.

This paper measures the impact of compiler optimizations by manually implementing individual optimizations and writing benchmarks in inline assembly code. The Rust compiler adds special annotations to inline assembly to ensure it passes through compilation as-is, without any further optimizations on top, allowing full control over code generation and control flow. These benchmark functions are written in C-style to hook into the SP1 prover system. For an example of how to integrate raw assembly into SP1, see Appendix A.

Chapter 4

Strength Reduction

This section explores how the strength reduction of constant integer multiplications affects the proof performance characteristics of compiled binaries.

4.1 Benchmark Design

This section provides a quick overview of multiplication and explains how to build benchmarks that contain multiplication instructions without getting optimized away.

4.1.1 Strength Reduction Background

I define the **degree** of a multiplicaion to be the number of strength-reduced instructions produced by the compiler. For constant integer multiplications, the degree depends entirely on the constant multiplicand. A power-of-two multiplication can be implemented in a single shift, so this multiplicand has degree one. The table in Fig 4.1 gives a few examples of multiplication degrees. The middle column of the table gives a generic templace for the multiplication $y = c * x$, expressing the coefficient c in terms of power-of-two multiplications to illustrate how to strength reduce using shifts. The variables m and n here are placeholders for arbitrary integers. The right column gives a concrete example of each type of multiplication,

providing constants for the right-hand side of the middle column. For example, the right column of the degree-3 row instantiates the middle column with $m = 4$, $n = 2$.

Degree	Code Equivalent	Sample Multiplicand
1	$y = x * 2^m$	$y = 16x$
2	$y = x * 2^m - x$	$y = 15x$
3	$y = x * 2^m - x * 2^n$	$y = 12x$

Figure 4.1: Multiplication degree table

Though any arbitrary integral multiplicand can be represented with adds and shifts, the compiler only peforms strength reductions for degrees up to 3. The definition of multiplication degree offered in this section applies throughout the entire chapter, as multiplication degree is an important parameter for benchmarking.

4.1.2 Benchmark Implementation

To generate a benchmark that directly uses the RISC-V multiplication instruction, the compiler must be subverted. Even with all LLVM optimizations disabled (using the `opt-level` flag for the Rust compiler), the compiler will still strength reduce multiplications to shifts whenever possible. These optimizations happen in the `InstCombine` LLVM transformation, which applies even at the most basic optimization level.

In the benchmark, an accumulator variable is repeatedly multiplied by random temporary variables, implemented by hand as either a multiplication instruction or a series of shifts and adds. To prevent the prover from doing constant propagation on the multiplicands, each multiplication instruction is interspersed with an XOR. These break the associativity and commutativity properties of the multiplication, preventing shortcuts. A Python script is used to programatically generate these scrips with hundreds of thousands of multiplication instructions.

4.2 Benchmark Results

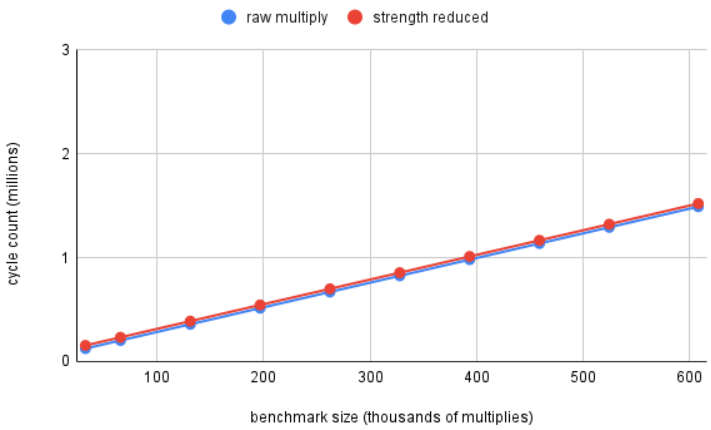
This section details the results for the generation of both the Core, sharded SP1 proofs and the user-friendly, smaller PLONK proofs given the same binaries. Binaries are evaluated for end-to-end prove time, execution cycle cost in the SP1 zkVM, and instruction count.

4.2.1 Core Proofs

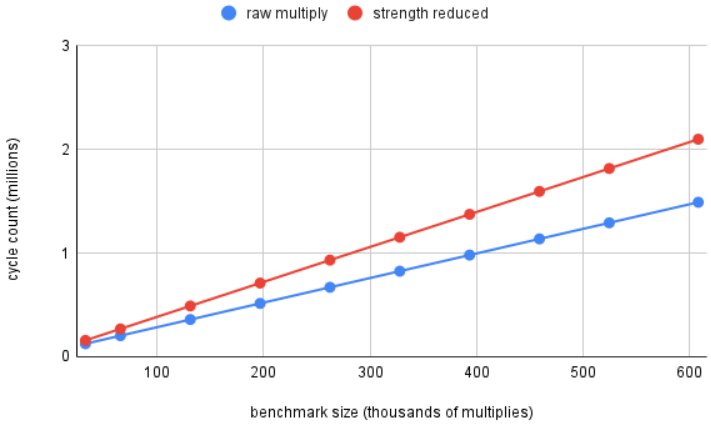
The results from this investigation suggest that strength reduction on high-degree multiplication has a negative performance impact on the generation of Core zk-STARK proofs. At the individual level, both instructions have comparable latencies. Fig. 4.2 shows the total executed cycles for each benchmark. The degree-1 case in graph (a) makes the parity clear: in this case, both binaries execute exactly one instruction per power-of-two multiplication and have the same total cost. Once additional instructions get compiled for higher-degree multiplications, the total cost increases even further.

The increase in the number of executed cycles appears to cause an increase in Core proof generation time, as seen in Fig. 4.3. This likely means that even if the prover chip for left shift is narrower than multiplication, the addition of more total height to the prover system causes a net increase in prover area and an increase in proof time. The effect is particularly pronoucned for the degree-3 case in graph (c), where prove times more than double for the largest benchmark sizes.

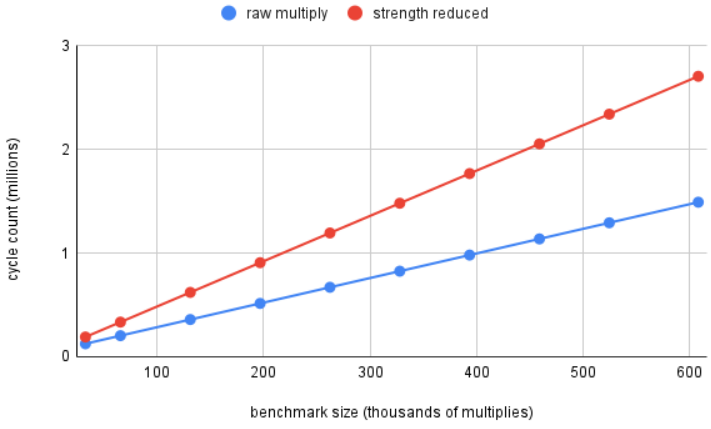
In summary, the data suggests that raw multiplications instructions are more efficient than strength reduction. The graph in Fig. 4.4 shows that despite sometimes requiring more execution time (the blue points tend more right), the multiplications require consistently less prove time relative to their execution cost. Since the graph axes are orders of magnitude apart, this graph makes the case that for zk-STARK proofs, the compiler is overfitting to CPU execution time by doing strength reduction optimizations. The only clear advantage of strength reduction is in the sharding. The strength reduced execution trace emit one less



(a) Cycles for degree-1 multiplicands

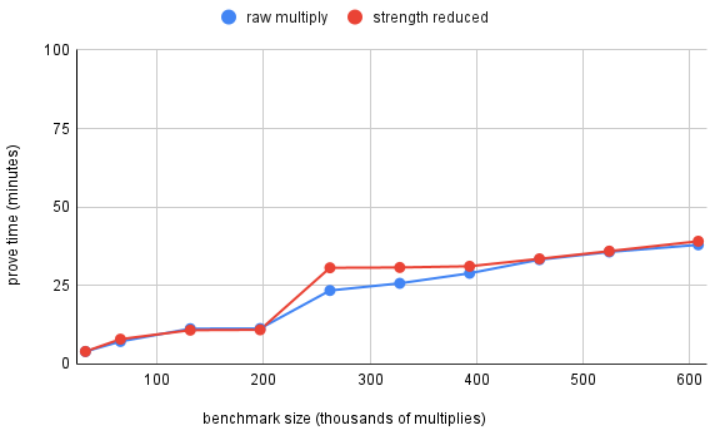


(b) Cycles for degree-2 multiplicands

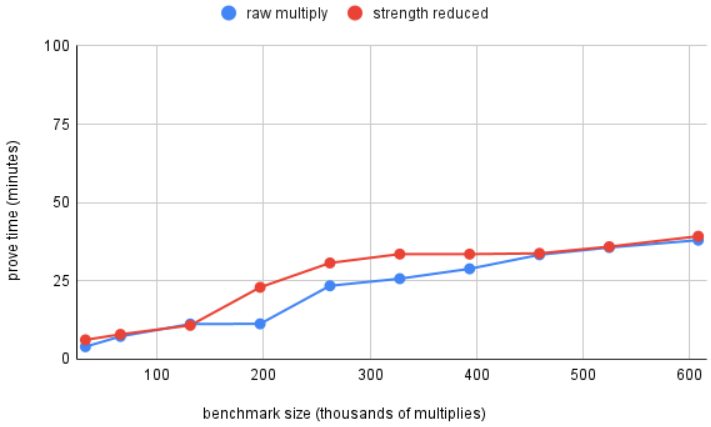


(c) Cycles for degree-3 multiplicands

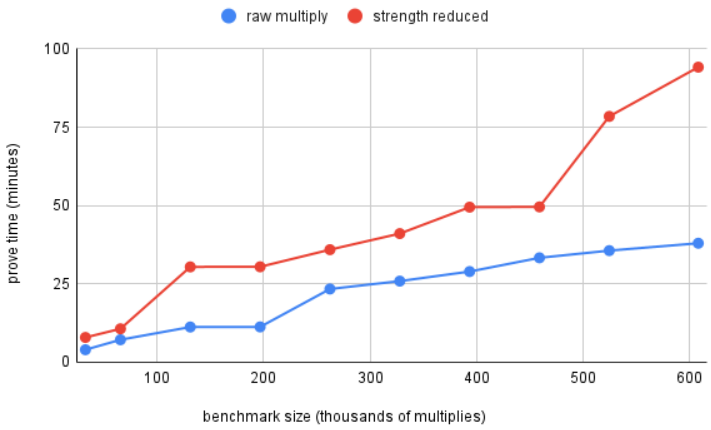
Figure 4.2: Executed zkVM cycles by benchmark size for various degrees



(a) Prove times for degree-1 multiplicands



(b) Prove times for degree-2 multiplicands



(c) Prove times for degree-3 multiplicands

Figure 4.3: zk-STARK proof generation time by benchmark size

shard for the degree-1 and degree-2 cases with more than 200,000 multiplications. This fact increases the size of the zk-STARK proof, but becomes a bigger issue for zk-SNARK proving in the next section.

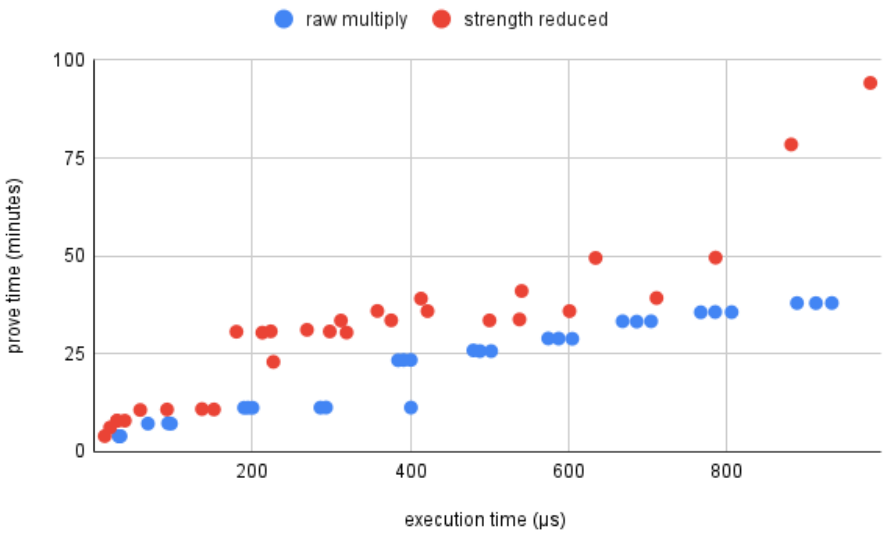


Figure 4.4: Prove-to-execute ratio for multiplications

4.2.2 PLONK Proofs

For this benchmark suite, differences in proof time come down to different durations of the `prove_core` zk-SNARK proof generation time and `compress` recursive proof phase; the remaining phases of the zk-SNARK proof generation were consistent between the optimized and unoptimized binaries. This reaffirms the findings from the degree-1 case in Section 4.2.1: at the individual instruction level, shift and multiply have comparable proof costs.

That section explores the difference in zk-STARK proof generation time that contribute to `prove_core` differences. To restate, strength reduction has a negative effect on these proof generation times, becoming more drastic as multiplicand degree increases.

The second source of proof time difference comes from the `compress` phase of the proof generation. As Section 2.5 explains, this part involves recursively calling the zk-STARK prover in a tree structure on the proof shards. The cost of this process is thus determined

by the number of shards produced from the zkVM execution trace. This is where strength reduction sees an advantage: even though strength reduction often produces binaries that cost more cycles to execute in the zkVM (Fig. 4.2), these larger binaries tend to shard less. This could be due to the increased width of the multiplication prover chip, or poor estimation by the sharding code. Further investigation is needed to understand why the number of shards differs between versions.

In general, the effect of strength reduction on zk-SNARK proof generation is the cost of extra zk-STARK prove time in `prove_core` subtracted from the savings of less recursive shard combinations in `compress`. The graphs in Fig. 4.6 visualize this combined effect.

4.3 Applications

Given the positive results found from the investigation, the next step would be to integrate the findings into more realistic code. Optimizing constant integer multiplications can already prove useful for calculating array access offsets based on dynamic data within a hot loop. The most basic example of this type of pattern is seen in a snippet like Fig. 4.5: the calculation of the outer array offset into `B` can be optimized.

```
1  char A[N];
2  char B[M][N];
3  int N;
4
5  char gen();
6
7  for (int i = 0; i < N; ++i) {
8    B[A[i]][i] = gen();
9  }
10
```

Figure 4.5: Example of optimizable code snippet

Using the compilation dataflow from Fig. 2.2, there are three possible approaches to undo the strength reduction

1. At **Rust source code**: any time that a degree-2 multiplication would be done, replace it with handwritten inline assembly

2. In the **Rust compile**: modify the compilation process such that compiled binaries no longer contain strength reductions

3. In the **RISC-V ELF**: transform the raw bytes of compiled binaries to undo the optimizations in-place

Manually editing all Rust source code before compilation is not desirable. For user-written code, this solution does not scale well and is prone to bugs. De-optimizing external dependencies would require forking libraries or convincing maintainers to publish alternate versions. Neither approach is very scalable.

The Rust compiler does not expose an API to toggle specific LLVM passes. Even if it did, `InstCombine` is too broad a transformation to completely disable. It provides optimizations like constant folding and combination of redundant instructions that would benefit prove times by keeping execution traces short. This leaves two options for modifying the compiler:

- Intercept LLVM-IR in the compilation pipeline

- Disable strength reduction in `InstCombine` code of LLVM

The first option has the compiler generate LLVM-IR, intercepts it, modifies the code, and then links it back into the ELF binary. This relies on a wrapper script for the Rust compiler to capture the LLVM-IR and metadata for each compiled program dependency. I was able to successfully intercept and edit the LLVM code to prevent strength reduction by marking all multiplications as inline assembly: the compiler cannot make optimizations on inline assembly code. This code successfully compiled to object files using the `llc` LLVM compiler, but could not be linked back into Rust's `.rlib` archives for library packages. The modified object files were not discoverable by other programs, rendering this approach ineffective.

The second option involves editing the source code for the Rust compiler itself. At the time of writing, the current release of the SP1 toolchain depends on the `succinct-1.88.0` release of the SP1 fork of the Rust compiler. Within the fork, the file `InstCombineMulDivRem.cpp` has a function `visitMul` that implements multiplication strength reduction. Removing this code and forking the dependencies should build a Rust compiler which does not perform strength reduction. This discovery was made late in the research project, and there was not enough time to test the approach. Testing this new compiler is left as future work.

In-place binary editing was the approach taken for this paper. Since the SP1 prover generates statically linked binary objects, iterating over a single file and replacing all the strength reductions with multiplications would cover both user code and all external dependencies. I wrote a Python script to do this, using a state machine over the patterns multiplication degree table in Fig. 4.1 to edit the raw binary in place.

This script was tested on select program binaries from the Rust compiler benchmarking suite [7] with detected reduced multiplications: a raytracer, physics simulation, and svg parser. Unfortunately, none of these saw significant speedups after modification by the script. A few immediate explanations for the lack of speedup stand out:

1. Few multiplications are in a hot code block. There might be some savings, but the performance bottleneck is elsewhere

2. There are enough multiplications to achieve savings, but the script cannot detect them reduction. A single multiplication is compiled into multiple, non-adjacent instructions within the same code block and is not detected by the script.

3. There are enough multiplications and they are replced, but they are in a hot loop. Within a hot loop, a raw multiplication instrution is at its most efficient when the constant multiplicand can be loaded into a register outside the loop (hoisted). The script does not current implement this behavior because it only does local line replacements, and does not know which registers can be safely clobbered or where to do so.

Other potential optimization areas suffer from the same problem and from better competing solutions. In the domain of hashing, where integer multiplications are common, SP1 already provides precompiled prover circuits that perform better than raw multiplication. Smaller, multiplication-based hash functions like SDBM may see some benefits from this technique, but even optimized SDBM falls short of modern hash functions like Rust's native FXHash. As such, the application of this project's findings to real-world programs is left as future work.

(a) Prove times for degree-1 multiplicands

(b) Prove times for degree-2 multiplicands

(c) Prove times for degree-3 multiplicands

Figure 4.6: PLONK proof generation time by benchmark size; left column shows times for strength-reduced multiplies, right column shows raw multiplication instruction results

Chapter 5

Instruction Benchmarking

5.1 Benchmark Design

A naive approach to benchmarking would profile a test binary for each of the possible instructions in the instruction set. This immediately stands out as an inefficient design due to the inherent similarities between some instructions; for example, addition has a register-to-register (`add`) instruction and a register-to-immediate (`addi`) variant. Benchmarking both would be redundant. Likewise, each RISC-V instruction gets translated into an AIR equivalent when the execution trace is turned into a zk-STARK proof. The benchmark only needs to cover one representative sample from the arithmetic prover chips. The table in Fig. 5.1 lists which RISC-V instructions were chosen for each prover chip.

Prover Chip	RISC-V Instruction
AddSub	`addi`
Bitwise	`xori`
Mul	`mul`
DivRem	`div`
ShiftLeft	`slli`
ShiftRight	`srli`

Figure 5.1: Prover chips and their representative RISC-V instructions

Similar to the strength reduction benchmarks in Section 4.1.2, inline assembly is used

to maximally restrict compiler behavior. The assembly code itself is generated using a Python script. This script simply loads random values into registers and combines them into accumulators using the target instruction. Each line generated by the script combines the random value loaded into the `t0` with the running accumulator in `a0`.

5.2 Results

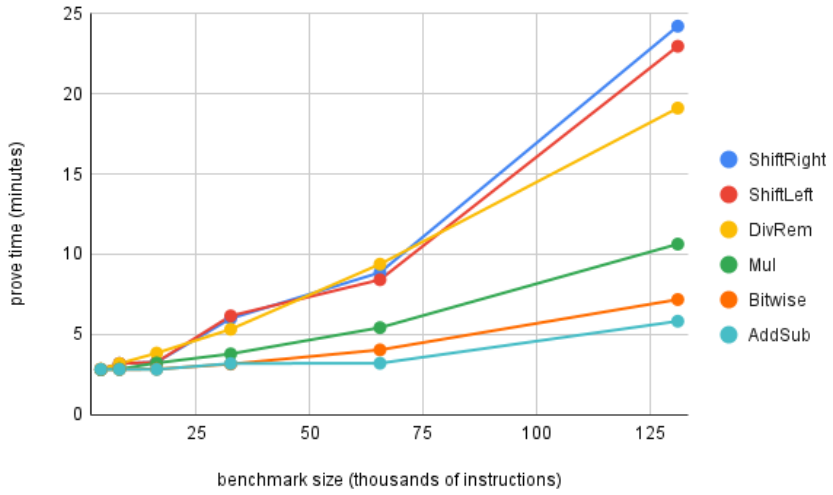
5.2.1 Exprimentation Bug

The results from the benchmarks are shown in Fig. 5.2. These results would suggest that bit-shifts are significantly more expensive than multiplication in the zkVM. However, notice that graph (b) shows a 4x gap in the number of instructions executed by the zkVM in some of the benchmarks. The ShiftLeft and ShiftRight chips are the only two plots colinear on the upper line; this suggests a bug in the benchmark design where the compiler added extra instructions and polluted the data collection. Investigating the execution report reveals that these extra instructions were `add`s and `lw`s.

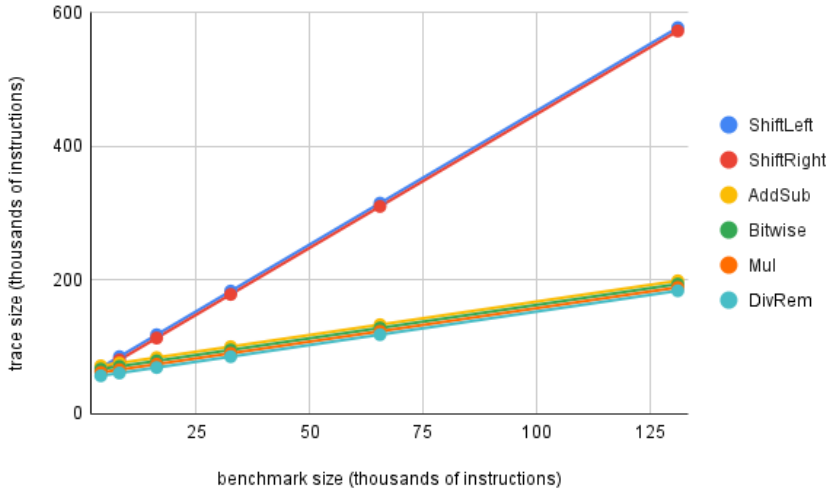
5.2.2 Adjusted Results

One possible approach to factor in this bug is to adjust the effective prover time for the ShiftLeft and ShiftRight prover chips. This can be done by dividing the marginal increases in prove time (difference between adjacent datapoints) by four. This division by itself is inaccurate - it ignores any constant overhead from the SP1 prover shared by all the binaries. Several observations about the graphs in Fig. 5.2 suggest that SP1 has a constant proof overhead:

- The plots in graph (b) exhibit a linear relationship with $R^2 = 1$. Despite different slopes, both lines intersect the vertical axis at approximately the same value, 47875 instructions.



(a) Prove costs for various prove chips



(b) Execution trace size for various prove chips

Figure 5.2: Results from buggy benchmark implementation

- The plots in graph (a) exhibit flatten out to approximately the same value, well above the horizontal axis.

- The plots in graph (a) have almost the same prove time for very small benchmark sizes.

These observations suggest that the constant overhead is approximately 170 seconds. At the smallest benchmark size, the test binaries contain 4096 deliberate test instructions. However, the correct benchmark binares execute 52000 instructions. The difference (SP1

overhead) would account for more than 10 times the number of target instruction executions, and should dominate proof time. At this smallest benchmark size, all the binaries prove in about 170 seconds.

The graph in Fig. 5.3 applies this adjustement to plot (a) from Fig. 5.2. This is only a rough approximation of what the data should look like, as it assumes that the extra prove compiler instructions have the same prove cost as a shift and that the SP1 imposes a constant 170 second overhead independent of the binary to prove.

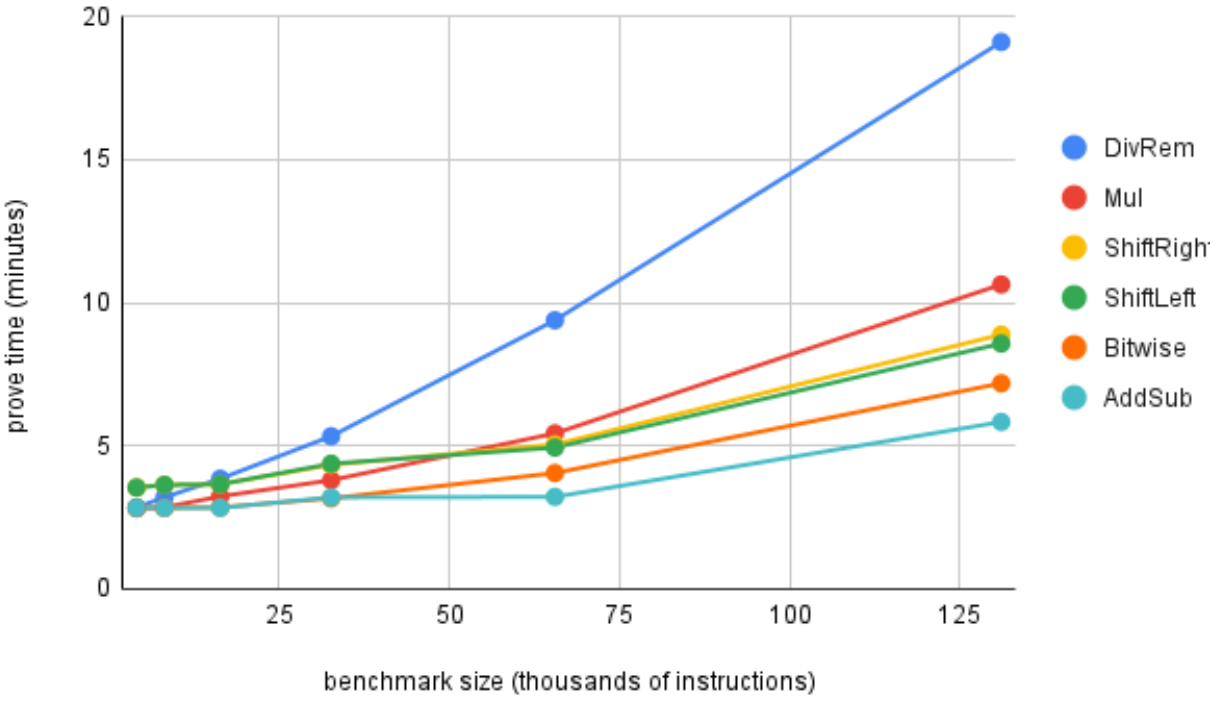


Figure 5.3: Prove costs for various prove chips, using the shift cost adjustment

The table in Fig. 5.4 normalizes the results into a relative cost index for each instruction. As expected, the cost of basic addition and subtraction arithmetic ranks lowest on this list. Bitwise logical operations and shift have comparable proof costs, with multiplication slightly higher. Fig. 5.5 plots the adjusted result of proof times compared to execution times for each chip. Divisions still pay an incredibly high prove cost relative to their execution time. Bitwise combinations instructions are slightly more efficient than both multiplication and

shift instructions.

Prover Chip	Relative Proof Cost
AddSub	1.00
Bitwise	1.51
ShiftLeft	1.28
ShiftRight	1.82
Mul	2.69
DivRem	5.61

Figure 5.4: Adjusted relative proof costs of each AIR chip

This adjusted result deviates slightly from the results of Chapter 4. That chapter finds that multiplications and shift instructions should have a much closer per-instruction proof cost, as evidenced by the similar performance of the degree-1 and multiplication cases. This table would suggest that multiplication has more prove cost than a combined addition and shift, which seems unlikely given the results of the degree-2 case. The degree-3 case does agree with these findings, as a multiplication has less prove cost than the combination of two left shifts and an addition.

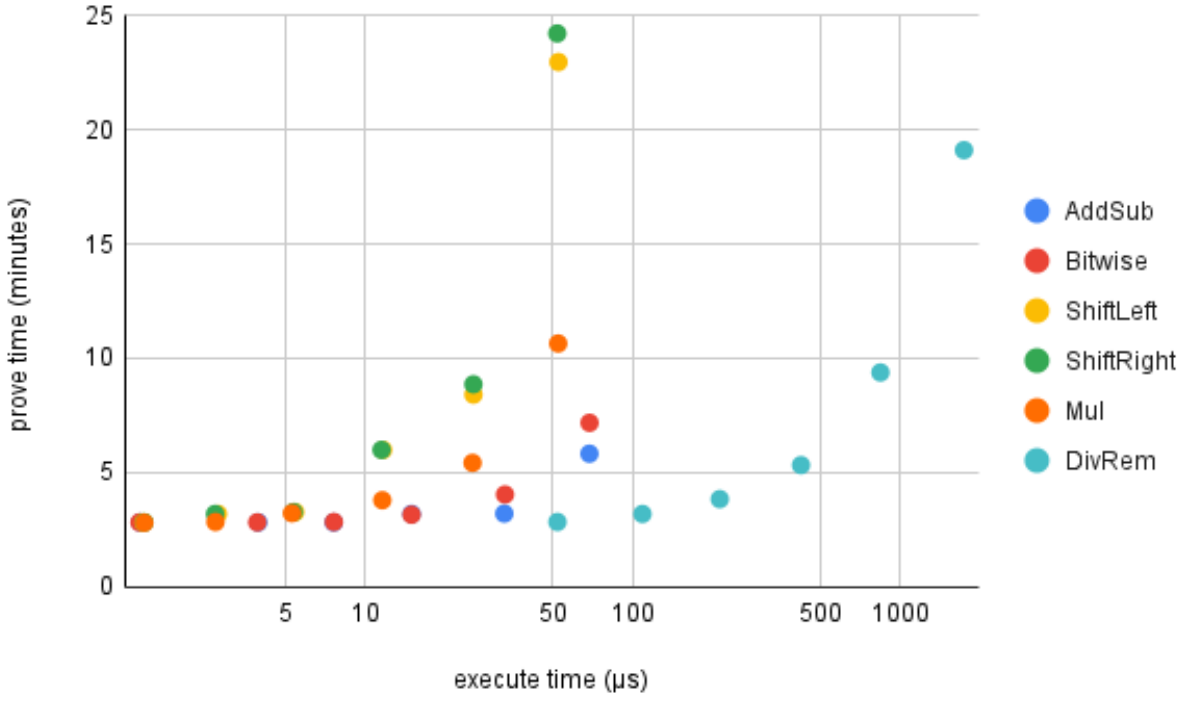


Figure 5.5: Prove-to-execute ratio for AIR chips

Chapter 6

Register Loading

Loading data into registers is a fundamental building block of any assembled program. A register can be loaded **explicitly** with a memory load (`lw`) or **implicitly** as the destination register of an operation (`add`, `xor`, etc). Fig 6.1 gives examples of both types of loads. This section explores the impact of this choice on end-to-end prove times.

```
.rodata
constant:
  .word 12345
.text
main:
  // implicit load
  addi t0, zero, 12345
  // explicit load
  la t2, constant
  lw t1, 0(t2)

```

Figure 6.1: Comparison of explicit and implicit register loads

6.1 Background

Modern compilers almost always prefer to do implicit loads whenever possible. In standard hardware, an implicit load costs less than 10 cycles.

By loading a register using an immediate instruction, an implicit load minimizes latency by executing an ALU operation in few cycles. Explicit loads can have significantly higher latencies. These must fetch data from memory addresses; if the target address is not in the cache, this instruction can spend tens or hundreds of cycles waiting for the data to be retrieved from main memory. Explicit loads also suffer from inducing more register pressure than implicit loads. When using explicit loads, compilers avoid this worst-case behavior through instruction-level parallelism, data prefetching, and better cache utilization patterns.

While implicit memory loads might have better worst-case performance, they are not always available to the compiler. Implicit loads can only be output when the desired data is known at compile-time and can be baked into the final binary. The particular implementation of these loads also depends on the size of the data in question. If the data uses up to 12 bits, a single-instruction implicit load (such as Fig. 6.1) can be used. Otherwise, an additional instruction (such as `lui`) is required to set the upper bits of the data. Aggressive use of implicit loads can be achived using staged compilation [8], where extra data can be provided at further stages of the compilation to insert into the final output binary.

The SP1 zkVM is not a standard execution environment. As a virtual machine, it does not suffer from the same worst-case performance as standard hardware. If the virtual memory allows for single-cycle explicit loads, it might be possible to get better end-to-end prove time by substituting for implicit loads. A single cycle would be preferable in the prove context for the following reasons:

- A single-instruction `lw` would have less latency than multi-cycle ALU instructions. This reduces the work done in the zkVM trace before Core proof generation
- An explicit load is guaranteed to be a single instruction, whereas implicit loads might need multiply (`lui` + `addi`) for large constants. Reducing the number of instructions reduces the size of the execution trace, minimizing the number of proof shards for `prove_core`

Even though these savings might be small for each individual instruction, they could add to a significant amount given how fundamental register loads are to most programs. The total amount of savings would be determined by the width for the prover chips of both instructions and the overhead of simulating virtual memory in the zkVM.

6.2 Benchmark Design

This section describes the design of a benchmark suite to measure the average cost of proving individual explicit and implicit memory loads.

6.2.1 Overview

Benchmarks are written in raw RISC-V using Rust's inline assembly macros. Each benchmark binary consists of many blocks, concatenated into a single function. There are no jump or branch instructions to avoid polluting the results.

Each block is a workflow with three steps:

1. Load values into registers
2. Combine values togther using a binary register-register operation
3. Write into an array of accumulator registers

To measure the impact of memory loads on the prove time, control the ratio of instruction loads (1) to useful instructions (2) is varied across benchmark binaries. The ratio of instructions to loads is the **degree** of a block; each benchmark binary is standardized such that all its blocks have the same degree.

6.2.2 Steiner Sets

The blocks must be designed to avoid repeating common subexpressions. These patterns allow the compiler to optimize using common subexpression elimination (CSE), moving common

subexpressions to their own temporary and propagating it in the code. The code in Fig. 6.2 gives an example of this kind of block. The subexpression $t0 + t1$ can be factored into its own temporary and added to both accumulators, removing one line of code. This problem only becomes more apparent as blocks grow larger and copies can propagate further.

```
benchmark:
  li t0, 0xCAB
  li t1, 0xFAD
  addi a0, t0
  addi a0, t1
  addi a1, t0
  addi a1, t1

```

Figure 6.2: Example of incorrect block implementation

Even if the compiler cannot implement CSE (this can be forced using inline assembly), the prover might still be able to make optimizations under the hood. This problem is prevented entirely using a Steiner system to guarantee no common subexpressions between accumulator updates. A Steiner system $S(t, k, n)$ describes how to create k-size groups of from a set of n distinct points points such that the largest common subset between any two distinct groups has size t. The setting of $t = 2$ models the constraint from above: no pair of registers may be operands for multiple distinct accumulator updates.

6.2.3 Implementation

Each benchmark is parameterized by the total number of loads (l) and block degree (k). The Steiner constraint forces the indirect parameter of register overhead (n): this value measures both the size of the accumulator array and the number of loads per block. This means that the number of blocks (b) is given by the equation $l = nb$. The table in Fig. 6.3 gives the value of n for each of the possible settings of k. Note that this table implies $k > 4$ is impossible: RISC-V only provides 32 registers, and the benchmark requires $2n$ registes to run correctly. The largest case of $k = 4$ is achieveable using caller-saved registers.

Degree (k)	Registers (n)
1	1
2	2
3	6
4	13

Figure 6.3: Required registers by benchmark degree

The pseudocode snippet in Fig. 6.4 gives an example of a single block with degree $k = 3$. Note the use of the `t`-series registers for loading temporaries and `a`-series as accumulators. Longer benchmark functions are be programatically generated as concatenations of these blocks using a Python script.

```
benchmark:
  li t0
  li t1
     ...
  li t5
  addi a0, a0, t0
  addi a0, a0, t1
  addi a0, a0, t3
  addi a1, a1, t1
  addi a1, a1, t2
  addi a1, a1, t4
    ...
  addi a5, a5, t5
  addi a5, a5, t0
  addi a5, a5, t2

```

Figure 6.4: Pseudocode for block with $k = 3$, $n = 6$

6.3 Results

The data from this investigation proved inconclusive. As expected, the zkVM's memory system saves cycles on explicit loads over register operations. The graphs in Fig. 6.5 show this effect.

Each graph has the total number of executed combination instructions bk on the horizontal

axis. Subtracting the $k = 2$ cycle count from the $k = 1$ count and dividing by half the benchmark size yields the average cycle cost per load instruction in the zkVM. This comes out to 1.5 cycles per implicit load and 1 cycle per explicit load. The result suggests that explicit loading always saves zkVM execution cycles over implicit loading.

The savings in executed zkVM cycles do not cleanly equate to savings in proof generation time. The graphs in Fig. 6.6 show the total proof time for a binary executing bk combination instructions.

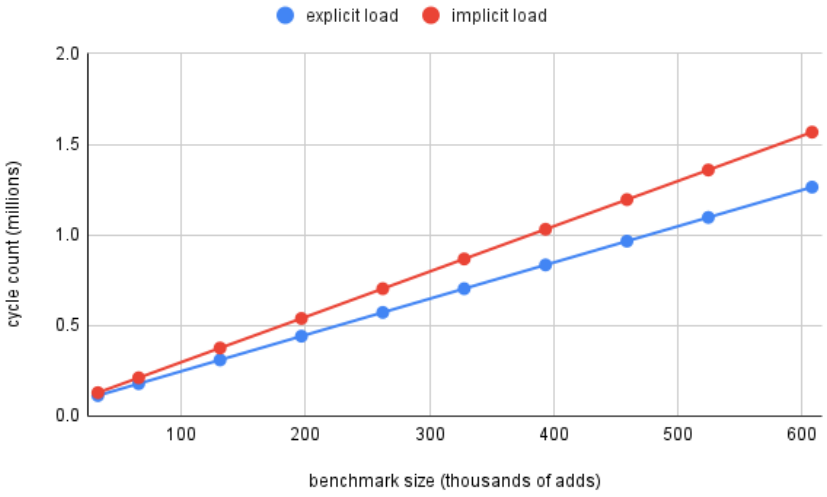
The results from Fig. 6.5 would suggest that explicit load instructions should see shorter prove times than implicit load instructions; this is not the case. This can be verified by measuring the average proof time per marginal load instruction for each benchmark size. For each benchmark executing c combination instructions:

1. Find the marginal prove time increase by subtracting the prove time in the $k = 2$ case from the $k = 1$ case.

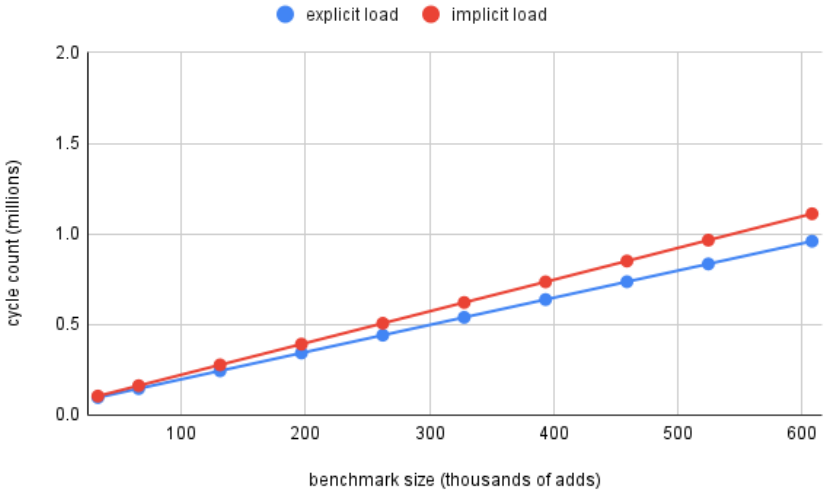
2. Find the total number of additional loads between the degree cases. Since a $k = 2$ benchmark does 2 combination instructions per load, this value is simply $c/2$.

3. Divide the value from (1) by the value from (2) to get the average proof time per marginal load.

The graph in Fig: 6.7 plots this value for each of the benchmark sizes c. If one plot were consistenly higher than the other, that memory load technique would by superior for proving. Unfortunately, no clear trend is observable from the graph.

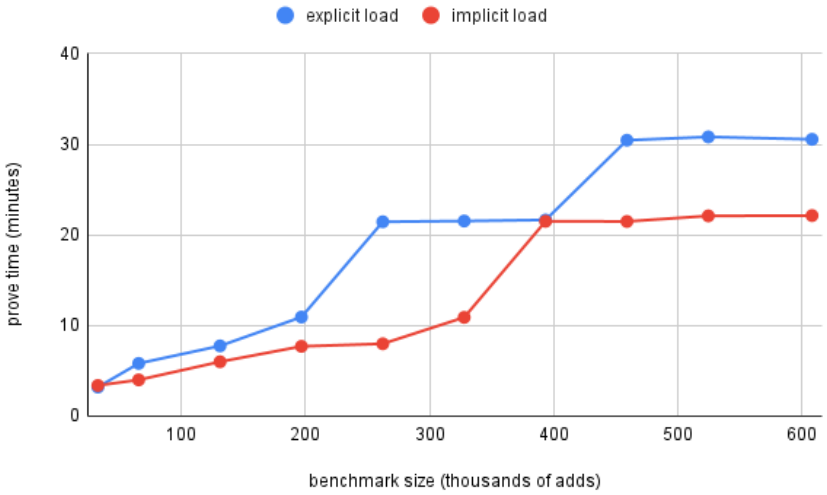


(a) Executed cycles for $k = 1$

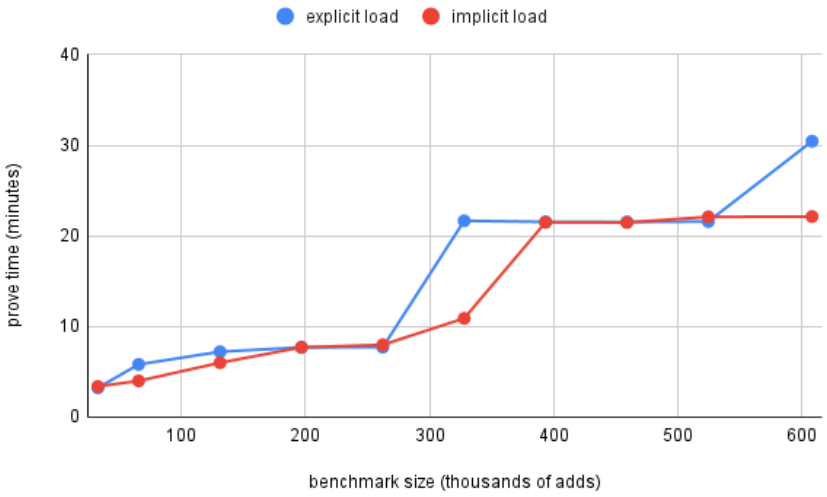


(b) Executed cycles for $k = 2$

Figure 6.5: Cycle cost for SP1 zkVM execution



(a) Prove time for $k = 1$



(b) Prove time for $k = 2$

Figure 6.6: End-to-end Core proof generation time

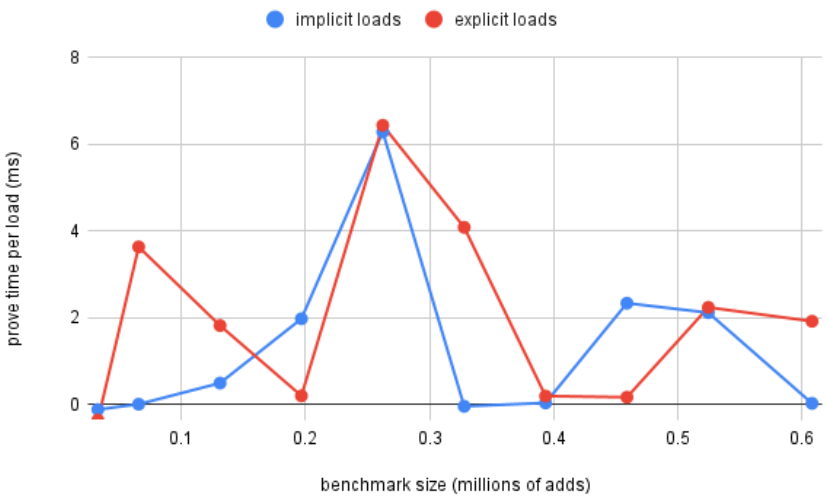


Figure 6.7: Average prove time per load

Chapter 7

Conclusion and Future Work

7.1 Future Work

7.1.1 Benchmarking Regular Execution

This paper did not benchmark execution of the RISC-V ELF binaries in a standard execution environment - either on physical hardware or through virtualization. Benchmarking in a regular execution environment would allow for a more nuanced evaluation of the results in this paper. As discussed in Chapter 5, the generation of a proof only needs to happen when a binary is distributed. The only recurring cost to a user of the binary is proof verification, which is designed to the fast. If the prove cost of a program (or even of individual instructions) can be weighed against the execution time, this would allow the user and developer to make more meaningful tradeoffs and maximize efficiency of their workflows.

7.1.2 Benchmarking a Custom Compiler

Chapter 4 found a positive result in the detrimental impact of strength reduction optimizations for constant integer multiplication, but did not find any compelling practical uses for it. Section 4.3 describes a process to build a custom Rust compiler from source that does not do strength reduction. Building this compiler would allow for direct A/B testing against the

standard compiler from the SP1 toolchain on production binaries. This approach should yield better results than the Python script, as instruction to load the multiplication coefficient can be re-ordered within the compiled code to maximize efficiency (e.g. through loop hoisting).

7.1.3 Further Register Benchmarking

The results from Chapter 6 were inconclusive as to the comparison between explicit and implicit data loads into registers. More targeted benchmarking might yield a more specific result to this question. The benchmarks in this paper only deal with loads of 12-bit immediate values. Implicit loads of immediate values with more than 12 bits would require more instructions to function, as large values cannot fit in the immediate `addi` or `xori` instructions used to implement the `li` pseudoinstructions. These extra instructions could certainly contribute to an increase in prove time over small, single-instruction loads. If a program was known to only use these large immediates (for example, when loading floats or strings), a definitive result for prover performance would be helpful.

7.2 Conclusion

In this paper I have shown how the compiler tends to overfit optimizations for the CPU computing environment. When these assumptions clash with the needs of the prover architecture, the overwhelming costs of code proof time create strong performance bottlenecks. This finding suggests that more work is needed in tailoring compilers to the specific domain of code proofs.

Appendix A

Sample Benchmark

```
#![no_main]
sp1_zkvm::entrypoint!(main);

global_asm!(r#''
.globl benchmark
.type benchmark, @function
benchmark:
  li a0, 0xCAB
  ret
``#);

extern ``C'' {
    fn benchmark() -> u32;
}

fn main() {
  let result = unsafe {
    benchmark()
  };
  sp1_zkvm::io::commit(&result);
}
```

References

[1] *SP1*. https://github.com/succinctlabs/sp1. Succinct Labs, 2025.

[2] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. Cryptology ePrint Archive, Paper 2014/349. 2014. URL: https://eprint.iacr.org/2014/349.

[3] A. M. Pinto. "An Introduction to the Use of zk-SNARKs in Blockchains." In: *Mathematical Research for Blockchain Economy*. Ed. by P. Pardalos, I. Kotsireas, Y. Guo, and W. Knottenbelt. Cham: Springer International Publishing, 2020, pp. 233–249. ISBN: 978-3-030-37110-4.

[4] *Plonky3: A toolkit for polynomial IOPs (PIOPs)*. https://github.com/Plonky3/Plonky3. Polygon Labs, 2025.

[5] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[6] S. Bowe, J. Grigg, and D. Hopwood. *Recursive Proof Composition without a Trusted Setup*. Cryptology ePrint Archive, Paper 2019/1021. 2019. URL: https://eprint.iacr.org/2019/1021.

[7] *rustc-perf: Website for graphing the performance of rustc*. https://github.com/rust-lang/rustc-perf. The Rust Foundation, 2025.

[8] A. Brahmakshatriya and S. Amarasinghe. "BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 39–51. DOI: 10.1109/CGO51591.2021.9370333.