Succinct Bootcamp: Recursion in SP1

Lecture Notes

September 26, 2024

1 Introduction to Recursion in Zero-Knowledge Proofs

1.1 What is Recursion in ZK?

In the context of Zero-Knowledge (ZK) proofs, recursion refers to the verification of one or more proofs inside another proof. This technique offers significant advantages:

- Reduces the overall size-cost of the proofs
- Decreases the cost (in terms of operations) of verifying the proofs

For instance, you can get one proof for the "price" of three, effectively reducing the verification cost to approximately one-third.

Reduced Size & Verification Cost

Figure 1: Concept of Recursion in ZK Proofs

2 Recursion in SP1

2.1 SP1's Approach to Recursion

SP1 employs recursion by splitting computations into "shards", each proving 1 million cycles of computation. These individual proofs are then recursed into a single validity proof.

2.2 Key Constraints in SP1 Recursion

1. Proper transition between shards
2. Global consistency across all shards
3. Appropriate use and access of the memory BUS for cross-shard communication

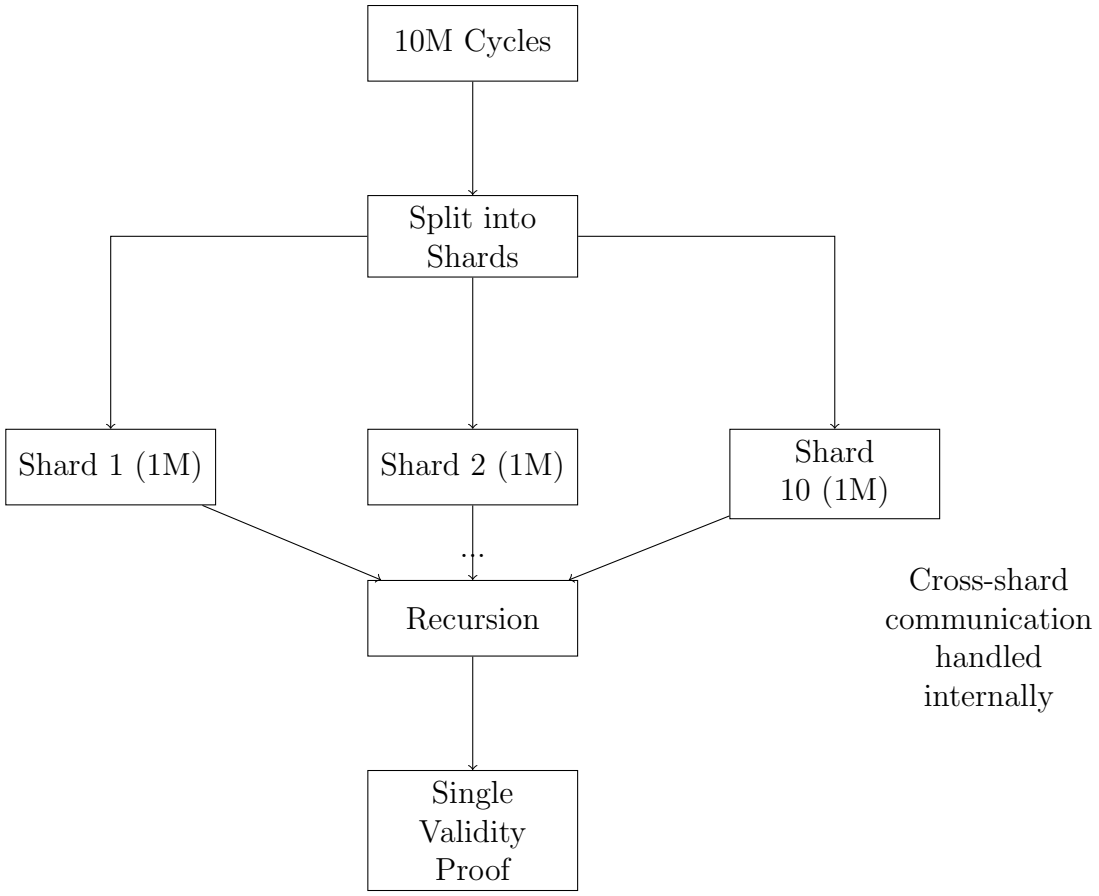

Figure 2: Simplified SP1 Recursion Process

3 Implementation Approaches

SP1 considered two main approaches for implementing recursion:

3.1 Option 1: Verify STARKs inside existing SP1 RISC-V

This approach was found to have enormous verification overhead, making it impractical. The overhead of the RISC-V encoding for the second verification procedure was too high, even with precompile acceleration attempts.

3.2 Option 2: Verify STARK inside a custom Recursion VM

This option optimizes the process by focusing on finite field operations and hashing. SP1 ultimately chose this approach. The custom Recursion VM has an Instruction Set Architecture (ISA) specifically designed for proof verification, including operations like field addition and efficient hashing.

4 SP1 Recursion Implementation (Version 1)

The initial implementation followed these steps:

1. Write a Rust DSL (Domain-Specific Language)
2. Compile to a higher-level Recursion IR (Intermediate Representation, similar to LLVM)
3. Transpile to a Recursion VM ISA (Instruction Set Architecture)
4. Optionally transpile to Groth16/Plonk ISA

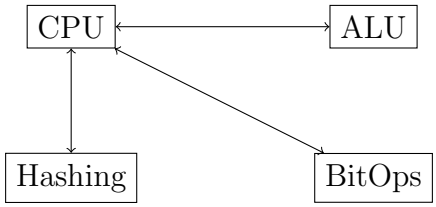

Figure 3: Recursion VM Architecture (Version 1)

4.1 Issues with Version 1

This version was eventually deprecated due to several issues:

- Loops had significant overhead
- AIR table structure was dynamic based on the computation being performed
- The compiler couldn't optimize the program effectively due to these dynamic structures

5 Improved Recursion Implementation

To address the issues in Version 1, SP1 developed a system of precompiled circuits:

- Multiple circuits (around 100,000) are pre-compiled for different shapes
- During recursion, the system selects the most appropriate circuit based on the proof shape
- This approach minimizes padding and optimizes performance

5.1 Circuit Selection Process

1. The system calculates the smallest distance from the current shape to any of the supported shapes. 2. It selects the closest shape that covers (is big enough to prove) the required computation. 3. If necessary, multiple normalization steps are performed to reach a standardized shape.

6 Stages of Recursion in SP1

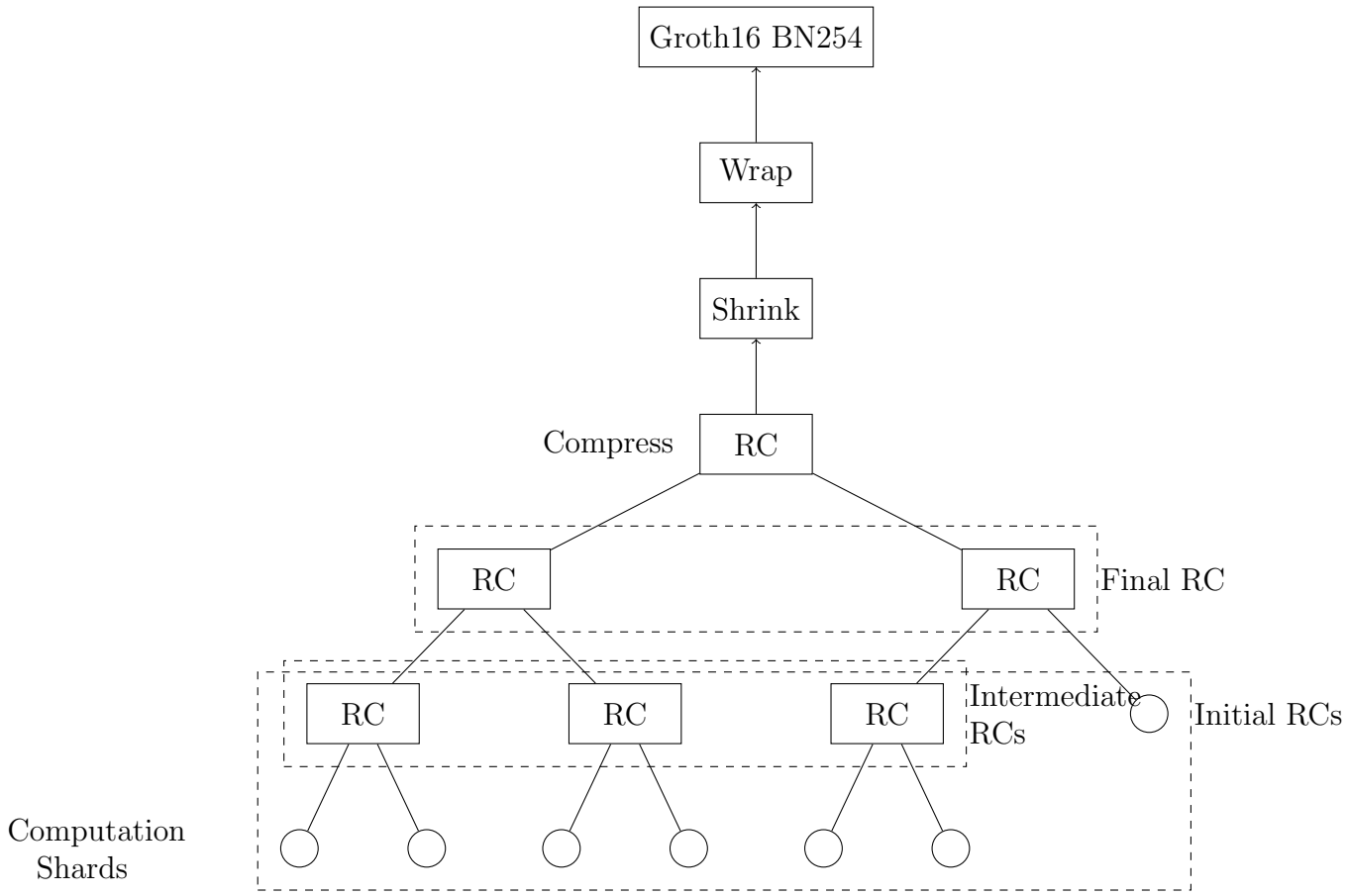

Figure 4: SP1 Recursive Proof Structure

6.1 1. Compress

- Normalizes shapes and compresses them
- Outputs a constant-size STARK Proof using all the execution shards (but it's still too big)
- Takes about 600 milliseconds to 2 seconds on GPU

6.2 2. Shrink

- Produces a constant-size STARK that is smaller than the compressed STARK
- Super cheap to verify, but uses the wrong field for blockchain use
- Takes about 1 second to compute

6.3 3. Wrap

- Constant-size STARK that changes the hashing function from Poseidon x BabyBear to Poseidon x BN254
- Compatible with Groth16 & Plonk
- Swaps out the field being used
- Takes about 2 seconds to compute

6.4 4. Groth16 BN254

- Verifies a STARK inside a SNARK
- Final stage of the recursion process
- Takes about 12 seconds to compute

7 Performance Considerations

- The core computation (generating initial proofs) takes about 5 seconds per shard
- The leaves (initial proofs) dominate the computation time, accounting for about 70 percent of the total time
- Parallelization can be applied to non-dependent parts of the process
- The minimum latency with full parallelization is determined by the cost of processing one shard, the number of layers in the recursion tree, and the fixed costs of the final stages

8 Conclusion

SP1's recursion implementation demonstrates a sophisticated approach to optimizing zero-knowledge proofs. By splitting computations into shards, using a custom VM, and employing a multi-stage recursion process, SP1 achieves efficient proof generation and verification. The use of precompiled circuits for different proof shapes further enhances performance and flexibility.

This approach positions SP1 as a powerful tool for generating efficient zero-knowledge proofs for complex computations, with potential applications in blockchain technology and other areas requiring secure, verifiable computation.