This is a continuation of Schemes for recursive proof composition and (N)IVC-based ZKVMs of the series “Compiling to (N)IVC ZKVMs”. Feedback is welcome

Motivation

While a ZKVM is not

a compiler, any end-to-end architecture for proving computation of a program likely involves the design of a compiler with that of a ZKVM. A ZKVM is often regarded as “something” that can prove the computation of a generic CPU program. Since a program is written in a high-level language, this seems to imply that a compiler is a component in a ZKVM, but it is not. For a given program, a compiler may output different circuits for the same instruction set of a ZKVM or, more interestingly, generate different instruction sets or ZKVMs that are optimised for that compiled program.

What is a program? One very common abstraction used in compilers is that of a control flow graph, which splits a program into a series of blocks and arrows of blocks based on jumps. A compiler takes a program written in a high-level language and outputs a circuit. The inputs and outputs of a circuit are finite field elements. A compiler should be able to \textit{decide}

which circuits or blocks are derived from a program and a prover would aim to minimise the cost of computing such subset of circuits that gets executed from a given input.

[

Screenshot 2024-01-18 at 12.42.05

928×528 45.4 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/30e1d946d5ea009f99851b4d79b722d4032fa366.png)

For example, given a program C={A, B, C, D, E, F, G}

and inputs x, w

for which the path P=[A, C, F]

is taken, a compiler will output the set of circuits while the prover’s costs will only depend on this path P

, not on the blocks not taken. With folding schemes, each path in P

can be folded into a single instance.

Different back-ends provide different trade-offs that will affect how the output of the compiler. This dance between compilers and proving systems will be the focus of this report.

In fact, we are not designing a ZKVM, but a compiler to a ZKVM. Whatever ZKVM will be suitable for a given program will be determined at compile time. Another way of seeing this, is that the set of instructions { F_1, ..., F_N}

of the ZKVM will be set \textit{dynamically}

at compile time. A question may arise: what is really a ZKVM? Does a different instruction set render a different ZKVM, or can the same ZKVM hold different instruction sets?

[

Screenshot 2024-01-18 at 12.40.58

960×522 46.7 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/2c46c886475924ab516d28b14aa7139bb68d7cb9.png)

We want to design a compiler that leverages the state-of-the-art work on folding schemes, thus outputting a NIVC-based ZKVM.

[

Screenshot 2024-01-18 at 12.42.58

1430×408 52.9 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/630e3b4ace91409a5f97382d5aa0bb4029883b11.png)

In the design of a ZKVM compiler, it is important to strike a balance between the size of each step function F

and the overhead induced by recursion, accumulation or folding. On one end of this spectrum, we have the whole program being one big circuit; on the other end, we fold of all primitive operations such as addition, equality or multiplication. There is no folding in the former case, rendering in the majority of cases a circuit that exceeds the current memory limits that a prover generally has access to. The base overhead of folding in the latter case may be comparable to the cost of proving due to having such small circuits (and there are many of them!). Adding a small folding overhead to many small circuits removes almost all the advantages of folding. So, where is the balance?

NIVC-based ZKVMs can benefit from compiler passes. While it is common to \textit{fix}

the set of functions {F_1, ..., F_n }

to a basic instruction set, it doesn't have to be so. A compiler can leverage the information it gathers from a given program to create these step functions F_i

at compile time. They no longer need to be small instructions, but subsets of the whole program created at compile time. Given some design constraints, the compiler, acting as a front-end to a SNARK, can split the program F

into a set of { F_1, ..., F_n }

subprograms. The compiler must be given a heuristic of this desired balance between circuit size and number of circuits. For example, equilibrium might be found by creating bounded circuits of 2^{12}

gates with only one witness column. The compiler must have an educated guess of how to decompose larger circuits into smaller ones.

This holistic ZKVM compiler approach is not commonly seen, since most projects are focused on building on top of fixed instruction sets (IS) such as RISC-V or one that is compatible with the EVM. However, we do find this integral frontend-backend approach in projects like Jolt and Lasso.

Example

Let's use an example to highlight the different ways of compiling a program. Let multi_algorithms

be a program that serves as a database of algorithms: given a program identifier, it proves knowledge of the outcome of that algorithm. We have mixed some algorithms together to illustrate how real-world applications tend to be assembled.

For example, in the first program (i.e. program_id=0

) we prove both knowledge of the n-th Fibonnacci number and that it is a prime number.

[

Screenshot 2024-01-29 at 19.49.56

976×752 73.5 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/fb75df293df30368857e0dc2622ebf45a8e3ee1a.png)

Let's revisit the three different compiling and proving approaches studied so far:

Monolithic circuits

This would be a direct compilation into a circuit for an arithmetisation (e.g. Halo2 Plonkish).

[

Screenshot 2024-01-29 at 11.26.32

902×324 6.54 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/89e94e4022dd75ba39214697a3b244bd604e0aee.png)

The circuit derived from this approach contains all the unused branches, since the compiler doesn't know the inputs of the program. For a complex application, this turns to be a large circuit with high memory requirements. Since, proving time is at least O(n \log n)

, where n

is the number of gates (including all the unused branches), this approach turns out to be highly inneficient for large applications.

Fixed instruction sets

This is the approach taken by STARK ZKVMs: the set of (primitive) instructions is pre-determined, that is, it is independent of the program. Underlying any computation on finite fields, there are primitives operations such as addition or multiplication, as well as other common and more complex operations such as range checks.

The prover doesn't prove the whole program at once, but proves each of these primitive operations, one at a time. In the case of STARKs, they use full recursion.

[

Screenshot 2024-01-29 at 11.48.32

1190×100 7.19 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/d721a1bb837471ffbfcb122198bb512b221c1999.png)

Having a set of fixed instructions for any possible program turns out to be quite inefficient when the prover provides a proof for each of these opcodes, since these opcodes happen to be too small to justify the overhead of proving.

Dynamic instruction sets

Fixed-instruction-set zkVMs don't have knowledge of the program

(so they include instructions that a particular program might never use).

In contrast, using some heuristic, a compiler may determine the set of instructions for a given program. For our example, a possible instruction set is depicted below. The aim of such compiler is to choose blocks of computation that are big enough to justify the overhead of folding and small enough to avoid blowing up memory resources and other issues derived from large circuits.

[

Screenshot 2024-01-26 at 17.54.24

2130×542 76.8 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/1da05135263e314356083d5881a0f72ff33324f9.png)

This set must be crafted with some considerations. Notice that functions such as range_check

, which checks that a value is between 0 and 2^n

, or is_prime

are independent of the rest of the computation. This means they can be treated as separate "opcodes" in this dynamic set of instructions. They are not too complex but not too small, and are used multiple times.

The different ways of splitting a program into subprograms that serve as an instruction set for a NIVC ZKVM is more an art than it is a science.

This post continues here