ZK-rollups boost transaction throughput in three ways:

(i) by moving contract state maintenance off-chain (only the contract's stateroot is maintained on-chain; that is, many individual updates to the contract's state are "rolled up" into a single update of the stateroot).

(ii) by moving signature checking off-chain (txn signatures are checked off-chain, as a zk-rollup snark proof is generated. The on-chain check is a batch verification that the snark proof is valid).

(iii) by moving contract logic off-chain (i.e. the contract's "business logic". The only on-chain logic is to verify that a new stateroot is valid, and to verify that new leaf data for the new stateroot is available).

Because one snark circuit is used for all purposes, if you want to use a zk-rollup to boost the throughput of your dapp, you have to implement all of your contract's logic in a snark circuit. Since implementing logic in a snark circuit is difficult, it's generally thought that zk-rollups are best applied applied to simple use cases, such as a contract that does token transfers and nothing more. The difficulty of implementing complex snark circuits is thought to be a blocker for widespread adoption of zk-rollups until "generalized snarks" are practical (meaning one universal snark circuit that would, for instance, generate a proof of correct execution of any EVM code that's passed as input).

But if most of the throughput boost achieved by a zk-rollup is due to (i) and (ii) and not (iii), then it should be fairly straightforward for any dapp, even if it has contracts with a lot of complex logic, to adopt a generic zk-rollup circuit for state maintenance - call it the zk-MerkleWitnessAndSigRollup or the zk-PenultimateRollup - and get a big scalability gain today (or as soon as someone implements it ;).

Perhaps the possibility of creating a generic zk-MerkleWitnessAndSig rollup has not been realized because contracts' state maintenance is often highly coupled to their business logic. But after some light refactoring to de-couple the business logic, a zk-MerkleWitnessAndSig rollup would be applicable (at least, such refactoring should be much easier than implementing the whole contract as a snark circuit).

So the question is: given a contract where the business logic and state maintenance are de-coupled, can a generic snark circuit be applied? I started wondering about this while prototyping stateless contracts for eth2/phase2, and seeing the prototypes of other Ewasm team members. Our prototypes tended to follow three steps:

1. verify input pre-state (i.e. the pre-state leaves), using witness data, against the pre-stateroot. And also verify signatures.

2. execute business logic on the input state to generate the post-state (i.e. the post-state leaves).

3. rehash the post-state to get the post-stateroot.

The witness data can be quite a lot of bytes. It's well known that snarks can verify merkle proofs (usually what we're referring to by "witness data"), so we can copy/paste some code from an existing snark circuit to handle step 1. The question restated is: how do we handle step 1 in the snark circuit, but handle steps 2 and 3 in the contract?

The issue is if we just do step 1 in the snark circuit, then when we go to do step 3 we have a problem, because we need most of the witness data from step 1 to rehash the state tree and get the post-stateroot. So we can't just do step 1 in the circuit (or we could, but we'd still need to pass in most of the witness data on-chain to the contract anyway).

New question: can we do steps 1 and 3 in the snark circuit, but leave step 2 implemented in the contract? This is where it gets tricky. A snark circuit that does step 1 takes two public inputs: the (hash of) the pre-state leaves, and the pre-state root (the witness data is taken as private input(s) - off-chain when the proof is generated). We could use this same snark circuit for both steps 1 and 3 in two different snark proofs: the first proof verifies that the pre-state leaves are included in a given pre-state root, and the second proof verifies that the post-state leaves are included in a given post-state root. But that wouldn't tell us that the given post-state root is correct (the post-state root might include other leaves that only the proof generator is aware of - this is the same "state poisoning attack" that the data availability guarantee of a zk-rollup protects against).

So we need to do steps 1 and 3 in the proof. The circuit would take four public inputs: pre-state account leafs (or the hash of the leafs), pre-state root, post-state account leafs (or the hash of the leafs), and post-state root. And the snark proof would verify that the only difference between the pre-state root and post-state root are the post-state account leafs that were passed in as a public input. In other words, the snark proof would verify that the post-state root is generated only from the state leaf updates that were the output of the contract business logic in step 2. Basically, the pre-state verification (verifying pre-state merkle proofs) and post-state update (insertions/modifications/deletions of tree nodes, and rehashing of the tree to get the post-state root) is done inside the snark circuit. But the post-state leafs themselves are computed outside of circuit (i.e. in the contract's Solidity/EVM/wasm logic).

Are there any issues with this approach?

I think this kind of snark circuit would be applicable as a common "generic merkle witness rollup" for a wide variety of contracts with complex business logic. And perhaps contracts based on this kind of circuit would also find it easier to interoperate with other contracts (e.g. an ERC20-compatible token that uses zk-rollups?), which seems difficult to do with existing zk-rollups.