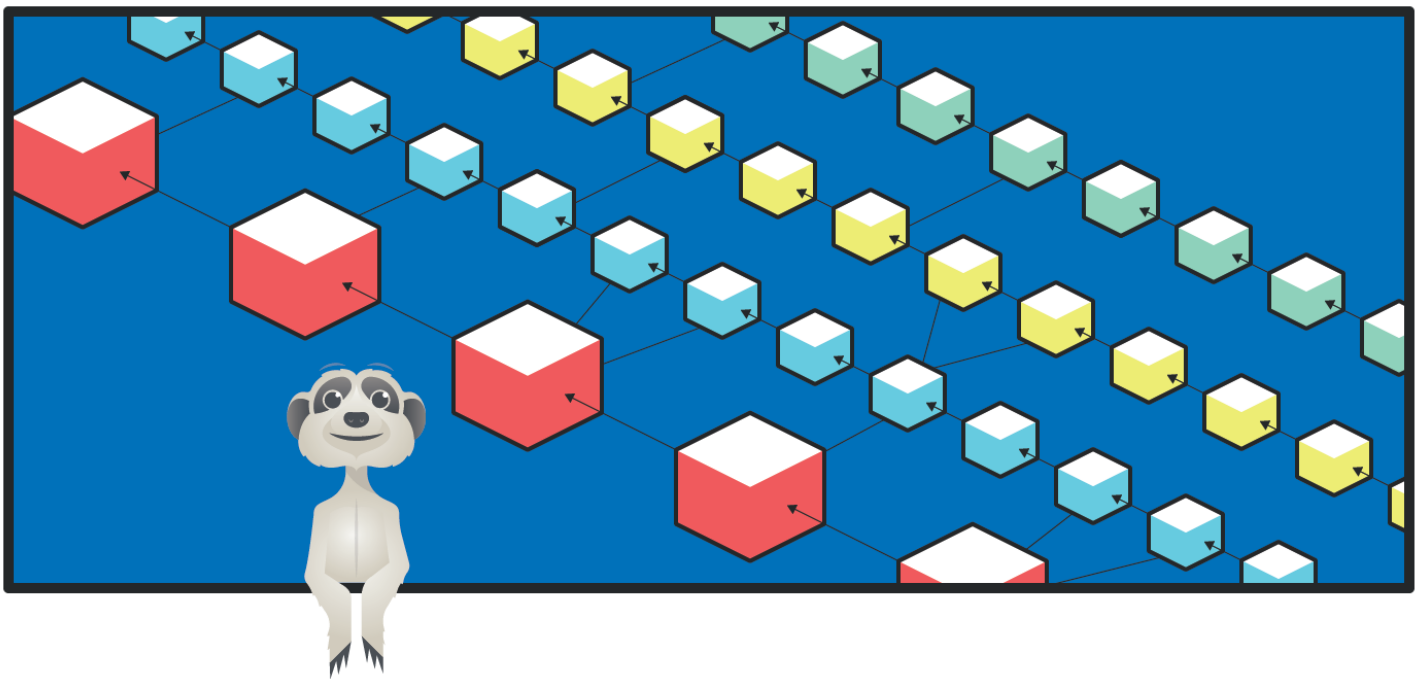
Wasm for Blockchain 2019

Maksym Zavershynskyi
Jul 11, 2019 · 21 min read

14 hours of content compressed into a 20 min read.



For those interested in Wasm and Blockchain but not fortunate to attend the workshop that happened in Berlin, I have summarized the content in this post.

eWASM ewasm Ewasm

Alex Beregszaszi @ Ewasm Team

Alex gave an in-depth walk through the story of Ewasm. The project started in 2015 with the search for a better VM for Ethereum. Back then they had an idea that blockchain will be running in a browser, or at least the light client will. So many candidates, like LLVM IR, were rejected because they did not optimize for portability. The choice fell on Wasm, but they wanted it to be fully compatible with EVM — support

the same features and allow contract interactions between Wasm and EVM. They also wanted to remove the need for EVM precompiles and have them implemented in Wasm.

By 2016 they had a metering implementation, EVM to Wasm compiler, JS implementation of Ewasm for browser, and C++ implementation. But in 2017 they hit the first blocker — compatibility with EVM requires Ewasm to be synchronous, however, EthereumJS (JS implementation of Ethereum) uses async methods, for things like storage interaction; also Wasm JS API for browser did not support async functions. They attempted to workaround sync/async incompatibility and as the result created Primea which was later adopted by Dfinity. Unfortunately, Primea has diverged too much from Ewasm and was no longer suitable for it.

After the project went to a stall in 2017 they have abandoned the in-browser idea in 2018 and went with sync approach. That's when they found about JIT bombs, and discovered one 300 bytes long (in Wasm bytecode) JIT bomb for Chrome, by exploiting V8 implementation specifics. They also decided to narrow down the scope of Ewasm to only precompiles which required integration with only 5 host functions instead of ~25. Also, they wanted to avoid metering injection in precompiles and use static analysis to estimate the cost; however, the halting problem prevented them from being able to do that. That made them reconsider interpreters instead of compilers.

In 2019 they discovered that highly-optimized EVM bytecode can reach close to native speeds. After they have implemented bignum library for 256 operations in Wasm, Wasm reached the speed close to EVM.

Focus today: Ewasm is now focused on interpreters and the minimal execution engine that uses only 5 host functions. The big question is cross-shard communication.

The Precompile Problem

Casey Detrio @ Ewasm Team

Casey's talk covered several related subjects: moving away from precompiles to Wasm; using compiled Wasm vs interpreted Wasm; and how his team is working on making interpreted Wasm as fast as EVM on crypto math.

Precompiles, in theory, is an EVM bytecode the implements some special cases and has a discount on gas cost; in practice it is implemented natively by Ethereum clients :) There are 8 precompiles currently in Ethereum clients, 4 of them were added at the

genesis, and 4 were added in October 2018 to support SNARK-related 128-bit operations.

Unfortunately, precompiles cause problems:

- They expand the trusted code base and make it more complex, gas rules for precompiles are hard to implement, and they introduce bugs;
- They require social consensus to be added, and each precompile encourages more lobbying;

Wasm might seem like a solution to the problem if they get it fast enough. Unfortunately, no Wasm engine allows 10-point pairings below the block gas limit; However, 2-point pairings are feasible. As discussed in the previous talk by Alex, they decide to go with interpreted Wasm as a stepping stone.

There is a debate on whether one should use compiled Wasm vs interpreted Wasm. In its current stage interpreted Wasm is not fast for crypto. Unfortunately, Wasm compilers are way too complex and have too many dependencies to pass social consensus to be merged into the codebase. Parity is working on a compiler that is potentially simple enough. On the other hand, Casey's team is working on improving the performance of Wasm interpreter. Out of the box, wabt and evmone have comparable performance on 64-bit arithmetics. With a ton of optimizations, wabt even gets close enough on 256-bit multiplication, however, it requires a bignum stack. A bignum stack is a virtual stack that is manually operated through host functions.

Focus today: Closing the gap between wabt and evmone+Weierstrudel on 128-bit multiplication. There is a debate on whether crypto is even should be in the focus for Wasm interpreters since most of the smart contract code is business logic. However, there is a huge demand in SNARK-like math and therefore they are currently focusing on it.

Is WebAssembly Suitable for Blockchain?

Paul Dworzanski @ Ewasm Team

Paul talks about various challenges of executing smart contracts on the blockchain.

The first challenge is determinism. For Wasm it is solved by avoiding floating point, being careful with the memory grow, careful with host functions, and with executors

implementations. It can be partially solved by having block proposer to execute all major implementations of Wasm executors and if there is a divergence drop the block.

Another challenge is the bombs. A bomb is input to Wasm engine that exhausts time or size. It is possible to have a 20kb code of nested loops that causes 2.5-sec compilation on Chrome's V8. It is also possible to use special shorthand notation in Wasm to expand 10 bytes of Wasm into 8GB of local variables.

The biggest challenge is metering. Unfortunately, gas metering injection purposefully impedes the execution optimizations, it is also often implemented through functions that do host calls. Fortunately, Ewasm created an optimization called "metering filtering" (a.k.a "superblock metering") that reduces the slowdown from 500x to 1.05–2.4x. Interestingly, using a handwritten metering function instead of a host call makes some improvement but in special cases. Also, there is an interesting idea to use exact metering formulas for certain functions, like keccak256.

Regarding the bytecode size, if we use blake2b as a benchmark, Wasm is comparable or beats EVM, depending on the level of optimization. Yay!

Focus today: Currently, the largest recognized bottleneck is not the business logic, but the crypto, especially SNARK-related math. See Alex's talk on how it can be improved using bignum API.

Ethereum 2.0 Execution

Casey Detrio and Alex Beregszaszi @ Ewasm team

Casey and Alex talked about Ethereum development phases, complex decision-making process between research and runtime teams, and the high-level concept of Execution Environment.

The design of Ethereum 2.0 implies the following separation of what needs to be agreed on:

1. The order of the transactions;
2. The result of the executed transactions.

Unfortunately, by spring 2019 the research is still stabilizing and it is not clear how protocol implements (2), which is a blocker for the design of the runtime. So the

runtime team is left with the following unanswered questions:

- Are we going with stateful contracts or stateless contracts? Stateful contracts need state rent and are harder to implement, but are easier to think of. Stateless execution has benefits of not needed to have a wallet online all the time;
- Immediate or delayed execution of the contracts?
- Will executors be different from validators?
- How cross-shard calls work?

Additionally, Ewasm team realized that there is no order on how cross-shard messages are delivered.

The unanswered questions and the realization about cross-shard messages led Ewasm team to decide to narrow down the scope, in which they ignore how (1) and (2) are separated and how exactly the order of transactions are agreed on. In this new scope, the executors receive blobs of data (a higher-level abstraction than a more specific set of transactions), which they run through Execution Environment (a higher-level abstraction than a specific EVM or Wasm virtual machines).

This Execution Environment could have the following specializations:

- Ethereum 1.0 block validator;
- Pure Wasm EE;
- Pure Wasm EE with cross-shard communication.
- SNARKs/STARKs verification;
- Various witness encoding needed for stateless execution, like SSZ partials;

These Execution Environments would be running on Wasm, therefore, we are going to have EVM running inside Wasm, Wasm running inside Wasm, etc. The Execution Environment Interface then would use five host functions that operate with blobs of data which can be considered as a low-level interface available to contracts running on pure Wasm EE. The Ethereum Environment Interface then would also provide ~33 high-level functions also available to the contracts through Rust API.

Focus today: Ewasm team has implemented the prototype of the EE, called Scout, see: https://github.com/ewasm/scout Scout defines Execution Environment Interface, testing format and tooling, provides example EE, and provides EE benchmarking; and it is only 400 LOC in Rust! To finalize this work Ewasm teams need to close the following questions:

- Execution Environment Interface;
- in-shard ETH transfers;
- in-shard calls;
- cross-shard calls;

Optional, desirable things to resolve:

- Fees;
- Relayer design — relayers are the ones relaying transactions from the users to the validators/executors.

WebAssembly System Interface

Lin Clark and Till Schneidereit @ Mozilla

The talk of Lin and Till largely follows the WASI announcement blog post but also gives some additional info about blockchain applications and updates on the new changes in WASI development directions. This summary will focus on the things not covered in the original blog post, so you will need to read it first if you are not familiar with WASI.

Now that we have a proliferation of Wasm runtimes for running Wasm outside the browser, Wasm team is afraid of digging themselves into the same hole that Node did. Node allowed running JS outside the browser, but in many cases, you would still need native modules for your out-of-browser applications either because they perform better than non-native counterparts or because you have code in other languages that you want to reuse. Unfortunately, native modules are not portable or secure and so Node made a decision early on to allow non-native JS modules to also have access to the system interface, e.g. read and write files. As a result, now that we have a lot of code written for Node using libc-like interface it would be hard to revert back to a more conservative interface that would enable security and portability. The main purpose of

WASI is to prevent this from happening to Wasm, thus its strong stress on security and portability — two major goals emphasized in almost every post on WASI.

Another hole that Node dug itself into caused by Node being de-facto the only runtime to run JS outside the browser, and so when you write JS you know which APIs to use, which basically makes Node a standard for outside the browser JS. With WASI we want developers to know which APIs they can work with, but not because one runtime is dominant.

WASI enables multiple applications of Wasm outside the browser: CDNs, Serverless, Edge computing, Blockchains, IoT, portable CLI tools. For instance, Fastly CDN is serving a significant fraction of the world's web traffic and they are moving from serving simple files to running customer's code on potentially every single request. It is feasible for them to create and tear down an instance at every single request because it takes 60 microseconds overall where 30 microseconds are spent on instantiating the modules. For comparison, V8 takes about 5ms to instantiate a JS module. In addition, to be faster than JS, Wasm is more scalable. Fastly's runtime requires only a few Kb of memory overhead, while JS engine for comparison takes tens of Mb. Therefore Fastly can fit tens of thousands of simultaneously running programs in the same process as opposed to hundreds with JS.

WASI takes a modular approach to the interface. Initially, it started by introducing wasi-core module that would cover most of the POSIX. However it had to introduce other modules too, for instance, not all POSIX systems support process forking, so they have created a wasi-processes module specifically for systems that do support it. WASI wants to define object-based high-level interfaces that work with specific use-cases, like smart contracts. Then if a developer is using some modular API, they can have the same niche toolchain, like environments and debugging tools. You can also have runtimes that are tailored towards specific use cases, instead of a one-size-fits-all solution.

WASI team is also working on a runtime that supports WASI — WASMTime. They expect WASMTime to become the best runtime for anything: IoT devices, cloud, blockchain.

Focus today: Moving away from wasi-core by splitting it into separate modules: file system, random numbers, clock, etc. This will allow, for instance, IoT devices that do not support file system to work with random numbers.

WebAssembly Beyond Browser

Dan Gohman @ Mozilla

Dan gives an overview of WASI and its history, he then talks more about the module system and access control model.

As Lin and Till explained before WASI tries to build a complex but yet generic security model through the access control. Anything that Wasm module tries to access needs to be granted to it from outside, either by the host or another module. For instance, in WASI module cannot access directories through the filepaths, it needs to be somehow given the directory file descriptor. The directory handle can either come from the pre-open mechanism or it can come from opening additional directories within another directory. WASI however, will unlikely support the mechanism of merging access rights, for instance, if the module has read access to the directory `/A` and write access to the directory `/A/B` then it won't be able to merge these to handles into one with read/write access to `/A/B`. WASI then will apply the same access rights model to sockets, by treating URLs similarly to file paths. Surprisingly, WASI also plans to have access control for random numbers, clocks, proc_exit, etc.

Dan also stresses the importance of modularity in WASI. He confirms what Lin and Till said, that wasi-core is going to be split into multiple modules. In general WASI APIs are just imports, and they don't have to come from the host, but can come from another module. So if the host does not provide access to random numbers, a module can provide an implementation of pseudo-random numbers to its submodules. This functionality of substitution is not implemented though.

Dan talks about the different categories of modules:

- Command. Does one thing — has `main` function and no other exports. Lifetime ends when `main` exits;
- Reactor. Does not have a `main` function, persists once setup function is called. Has an export API. Very useful for async I/O;
- Library.

This categorization will make the usage of modules intuitively easy. Right now Wasm toolchain supports command modules only.

Dan also talks about standardizing blockchain API through WASI, however, he recognizes that blockchain API is currently too diverse, so it might be possible to standardize only a subset of it, e.g. KV storage. The cool thing about WASI modules is that we can iterate on them. For instance, we can come up with some `wasi-kv` module for now and later if we decide to change the API we can create another module with a different name, e.g. `wasi-kv2`, and make `wasi-kv` a library implemented on the top of `wasi-kv2`. This gives us a lot of confidence in trying to standardize APIs. One of the audience members though expresses concern that even KV storage might not be standard enough for blockchains since some of them attempt to operate on memory pages instead.

Focus today: Stress on modularity and portability. Unify sockets, files, and other streamy things. Positioning in files is not generalizable to sockets though, more work needs to be done.

Designing a WebAssembly Runtime for Blockchains

Syrus Akbary @ WASMER.io

Syrus gives a brief overview of Wasmer and then focuses on blockchain specific features, like metering and blockchain API standardization.

Wasmer is a wrapper around several Wasm backends: Cranelift, LLVM, and Singlepass. Together these backends offer a tradeoff between compilation time and runtime speed, with Singlepass having the fastest compilation and LLVM having the fastest runtime speed due to numerous optimizations. Wasmer also provides embeddings for using Wasm backends in languages like Go (used by Perlin, Cosmos, Ethereum), Rust (used by Polkadot, EVM), Python, Ruby, C/C++, and Java coming soon.

Wasmer has been working on improving the metering speed (see Ewasm talk about it too). The difficulty is that the metering should be deterministic regardless of the backend. Today, the standard approach to metering is the one implemented by Parity where they transform Wasm module by injecting the calls to the gas meter on the host. Wasmer introduces a new level of abstraction — Middleware which injects the instructions at the parsing stage of the compiler. It also inlines metering more aggressively, instead of making a call to the host before every operation, their metering calls host only to get the gas threshold that it saves to a variable that it uses across the code, reducing the number of calls to the host. This results in reducing the metering overhead 8 times.

Wasmer also works on unifying APIs of Wasm modules. They have introduced Contracts (not to be confused with smart contracts) which is basically a set of imports and exports required in the module, represented in a Lisp-like format. The Contracts are planned to be extensively used in their package manager, WAPM, which by the way is going to be distributed through IPFS.

Wasmer opens discussion of unifying APIs of smart contracts, by presenting two approaches: the first one is the one suggested by Oasis in this blog post, where they emulate blockchain interface through POSIX-like imports available in standard WASI modules; the second approach is to either use imports without WASI (Ewasm approach) or have a special WASI module for blockchains. The audience leans towards the second approach, suggesting not to use POSIX where it is not strictly required, citing this WASI critique blog post by Ben Laurie. Till Schneidereit from WASI confirms that it does not make sense to emulate POSIX interface for WASI contracts but it might make sense to use the module system to define small use-case specific high-level API.

Focus today: Support for ARM. More language embeddings. Tiering feature — Wasmer should recognize hot functions and replace them with more optimized code.

The Blockchain Wasm Stack — Use-Cases and Requirements

Fredrik Harrysson @ Parity

Fredrik talks about how requirements on smart contract code and runtime shape certain decisions that they make in Parity. At the end of the talk he and audience debate on standardization of smart contract API.

Parity was the first to create a public permissionless blockchain running Wasm — Kovan. Unfortunately, no one actually used it, the major reason was the contract language, PWASM, being pretty bad. The end-users or even the contract developers do not care whether something is running on Wasm, but contract developers do care about the development experience. Learning from PWASM they have created Ink! — eDSL for smart contracts written in Rust. Parity considers Ink! to be a production-grade smart contract language, while PWASM was more like a proof-of-concept.

Parity is all about security first, which is why they went with Wasm interpreter rather than a compiler. In general, Wasm interpreters are pretty bad in matching EVM speed, but compilers have multiple problems: bombs, heavy dependencies, large attack

surface, different prioritization (Cranelift developers do not really prioritize determinism), etc. Compilers also create problems with the metering. Suppose we use a compiler and at some point, compiler update releases an optimization, do we reduce the gas cost for the corresponding operation? Changing gas costs is actually a huge security issue, as recently shown by Ethereum hard fork.

Fredrik also explores three dimensions of smart contract requirements: trusted vs untrusted code, code size, and speed. There are some challenges with these dimensions and Wasm contracts. It is really hard to trust user-uploaded Wasm code because it can contain bombs, but he suggests that even if we go with the compiler (rather than interpreter) there can be a game-theoretic mechanism where certain people vouch for the code to not contain a bomb and if it does they get slashed. He also supports Ewasm's idea of having pre-defined cost for precompiles to avoid metering.

Size is a huge issue for Wasm smart contracts. Ethereum 1.0 has a 24kb limit on the size and most of the Solidity contracts are within hundreds of bytes. On the other hand, Rust smart contracts compiled into Wasm are 10x of that. Currently, this is addressed by developing minifying tools, also it might be possible to deploy a standard library equivalent on the blockchain.

Speed is another challenge for Wasm if we want to match its speed with Geth. Geth uses some highly-optimized code, e.g. b128 library that Geth took from Cloudflare is half hand-written assembly tuned for x86–64. If we want to allow SNARKs and other heavy crypto on Wasm then things like hashing and elliptic curve pairing should be done in microseconds or nanoseconds.

Finally, Frederik and the audience talk about standardizing smart contract API. Fredrik is excited about having a smart-contract-specific WASI module but not sure it can be fully done within the next 20 years. Before that, we can have some part of the API standardized though. Some of his colleagues express skepticism on relying on WASI because they would need to ship before WASI reaches its maturity.

Lightbeam Wasm Compiler

Jack Fransham @ Parity

Jack gives a very low-level talk about the new streaming compiler from Parity. Many details related to specific low-level issues of Wasm and assembly are omitted in this summary.

In general, there is a clear preference of interpreters and streaming compilers over more complex compilers in the blockchain world. The problems with more complex compilers are numerous and were discussed in other talks too, other issues brought up by Jack are: complex compilers have long compilation time, which means users need to pay for it. However, then we also need a function to price compilation and if the function itself is complex we might need to price that function too.

On the other hand, with interpreter or streaming compiler we can execute Wasm immediately after we finished reading it. With the interpreter, it is achieved by executing Wasm directly. With the streaming compiler, we can emit assembly as we read Wasm and use it to execute the binary immediately. Interpreter, however, has multiple advantages: easy to build, easy to maintain, easy to debug, easy to ensure correctness, easy to write a simple API for end-users, can easily swap code at runtime, 100% memory safe, 100% thread safe, portable between platforms. The single major advantage of the streaming compiler is that it is fast. Comparing WASMI interpreter to Lightbeam we can see that it is approximately two orders of magnitude faster:

```
test native ... bench: 22,289 ns/iter (+/- 1,240)
test wasmi ... bench: 3,602,303 ns/iter (+/- 41,855)
test lightbeam ... bench: 22,266 ns/iter (+/- 1,242)
```

Notice that Lightbeam appears to be slightly faster than the native compiler, Jack's hypothesis is that it is because they have introduced an additional IR — Lightbeam IR — that allows more optimizations. With standard Wasm compilation the pipeline is the following:
Rust → MIR → LLVM IR → Wasm → Native

With Lightbeam the pipeline has additional step:
Rust → MIR → LLVM IR → Wasm → Lightbeam IR → Native

Lightbeam IR is a very simplified Wasm, basically, Lightbeam removes locals and hierarchical control flow from Wasm.

Current focus: Lightbeam seems to be passing all tests, but it does not mean it works. Performance improvements are possible. Needs fuzz testing for miscompilations and bombs. Adding support for 32 bit x86 and ARM64. Need to make it actually streaming as it is currently using some placeholders and backtracking.

An Awesome Runtime (OS)

Martin Becze

Martin's talk is a call to action to start building an OS that all blockchains can support and all smart contracts can run on.

Martin starts with reminding that one of the promises of the blockchain is portability which is currently left unfulfilled because blockchain contracts are not portable across the blockchains, which is primarily because we do not have standard interfaces and tooling. Ewasm was the first to take a stab into this problem, by standardizing interfaces and metering. Unfortunately, Ewasm was not designed to be portable, because they did not think of it as an operating system. However, any blockchain that has smart contracts talking to each other is implementing an OS. The blockchains then can be viewed as similar to the PC manufacturers — adapting to the OS layer. Such an OS layer should exercise POLA (Principle of Least Authority) and not use ambient authority, which is by the way not how current *nix or Ethereum work.

In general, OS needs performance, security, portability, and toolchain compatibility. But blockchain OS also needs determinism and integrity checks on the file system which is currently usually done through Merklization. It might actually be advantageous to have a Merklized file system when running such OS locally not on the blockchain. If we have that then smart could run on local machine and blockchain. Conversely, it would be nice if smart contracts had access to the POSIX-like API. If we could map smart contract filesystem to Linux filesystem without virtualization that would be great. This comes back to the discussion of WASI and POSIX that happened in other talks. Martin then opens discussion with the audience and the audience suggests that for now we can build adapters for POSIX but we should not constrain the development of an alternative standard that is not compromised by the legacy of POSIX; Martin agrees. The audience also notes that trying to have programs that run locally, on the cloud, and on the blockchain might be an overreach, because we are still not able to run the same programs on phones and cloud, which is the first problem we need to solve.

Current focus: Martin's talk is a call to action to start building interfaces, kernel, and tooling that would eventually lead to the OS.

Evolving Wasm into a Proper Misnomer

Andreas Rossberg @ Dfinity

Andreas talks about future Wasm features and the formalization aspect.

When developing new features Wasm tries to balance performance, simplicity, generality, and safety, where safety is achieved by having formalization of the entire Wasm which is used for machine verification with Isabelle, Coq, and K. Thanks to the initial strong focus on safety Wasm is easy to formalize and its formalization is about 2 pages long compared to 160 pages of JVM. Interestingly, JVM does not optimize for generality either because it looks like the language it was built for, while Wasm was not designed for any specific language.

Andreas then talks about highly anticipated future features of Wasm:

- **Multivalues** allow instructions and functions can have multiple results, e.g. division with remainder. The implementation of multivalues is basically about removing the existing restrictions from the formal form of Wasm;
- **Tail calls** are also highly anticipated and will be enabled by special instructions;
- **Bulk instructions** will allow efficient manipulation of data regions, akin to `memcopy`. Their implementation would require the references feature;
- **References** initially were a part of the GC proposal but then were split out so that they can be shipped faster than GC. The motivation is simple: there is currently no easy way to pass DOM object into Wasm and store a reference to it. References provide easy, efficient, safe interop with host environment; they would also set the stage for future features like exceptions and GC. There are two exception types that need to be added to Wasm: `anyref` — general reference, and its subtype — `funcref` — specialized function reference. Interestingly, this will also be the first instance of subtyping in Wasm. References would also be provided by the host only while Wasm code would only be able to store them and pass them back;
- **Exceptions**. Funnily enough, today if you compile C++ into Wasm all try blocks are going to be turned into JS calls, which is slow and too specific. It was not actually a problem for C++ projects because the majority of them were not using exceptions anyway :)
- **Stack switching** is needed for various control abstractions from other languages, like coroutines, continuations, lightweight threads, async/await, etc.

Implementation is basically an extension of the exceptions mechanism using "resumption";

- **Garbage collection**. It is kinda silly that many languages reimplement GC. Having GC in Wasm will solve that, but will also allow better interop with the host. However, it will be limited to low-level data-model like light structs and arrays, without full-blown objects. Unfortunately, while it simplifies certain things it will also create a minor overhead when using it.

Current focus: In addition to the above features, Wasm wants to standardize subsets of Wasm language because not everyone needs every feature, e.g. blockchain does not need threads.

Semantics of WebAssembly in the K Framework

Rikard Hjort @ RuntimeVerification.com

Rikard gives a hands-on workshop on using K framework.

K framework allows having runnable formal specifications of programming languages. Once you have a formal specification of the language you can generate things like: parser, interpreter, model checker, compiler, and test-cases. They have already provided parser, interpreter, debugger and reachability logic prover for real-life languages like Java, C11, KVM, LLVM; also in progress are Solidity and Rust. The project is very focused on the blockchains and they already have a formal spec for EVM in K, KEVM, accompanied by a repository of formally verified smart contracts, see: https://github.com/runtimeverification/verified-smart-contracts

Fast, Deterministic, and Verifiable Computations with WebAssembly

Mike Voronov @ Fluence

Mike talks about the Fluence blockchain that uses a variation of consensus that relies on a verification game, similar to Truebit. When there is a dispute over a certain state that nodes disagree on, the verification game allows to narrow down the difference to a single Wasm instruction. Then the dispute is resolved on Ethereum by rerunning the disputed instruction. They find the instruction that diverged using the k-ary search of the state trace. In order to execute the instruction on Ethereum, they need to send the proofs of memory, stack, instruction pointer, executed instructions counter, and used

gas compacted through hashing and Merlization. However, due to its size, Merklization of memory is tricky and so they only Merklize the “dirty” chunks of the memory, i.e. chunks that were modified. In turn, this requires charging users per chunk that is dirty to avoid grinding attacks that dirty the entire memory. Unfortunately, they do not support import from hosts at the moment but will allow a subset of WASI in the future.

Spacemesh smart contracts research

Yaron Wittenstein @ Spacemesh

Yaron presents Spacemesh which uses “mesh” instead of a blockchain and PoST instead of PoW/PoS. Spacemesh is considering three options for smart contract languages: Solidity (EVM running on Wasm), eDSL on Rust, and their own language SMESH. SMESH is going to be safe by design, will use ahead-of-time computation whenever possible, and will be targeting Wasm. They will use WASMTime and Wasmer+Cranelift to run Wasm contracts.

Blockchain Webassembly Rust Ethereum