We need a new, common Virtual Execution Environment

Herman Venter

Research in Software Engineering Group

Microsoft Research, Redmond

Why a VEE?

- I.e. why not just compile language X directly to a target chip architecture, perhaps using a common compiler back-end infrastructure?
- VEE provides greater platform independence by post-phoning the machine code generation.
- VEE factors out the TCB. No need to trust compiler for language X or to require compilation on client machine.

Why common?

- VEE requires big investment: 100+ person years
- Expected lifetime must be 20+ years
- Tool support
- Libraries
- Eco system

Why new?

- JVM has strong pro arguments
  - Ubiquitous
  - RVM
  - 200+ languages
  - J2EE, Eclipse, eco-system
- But
  - Designed for one language
  - Too complex, too much baggage
  - Extensions take time to introduce, break so many things and need so much effort to implement that in effect, one produces a new VEE
  - Not ubiquitous enough

New challenges

- Vector machines
- Many-core
- NUMA
- Grids and Clouds
- Versioning and long lived data formats

Can we realistically expect to do better with a new VEE?

- How does one design for multiple languages?
  - JVM has more languages targeting it than the CLR, whose designers consciously tried to accommodate multiple languages
- Wouldn't a multi language VEE necessarily be more complex than the JVM?
- If the JVM is not ubiquitous enough, why is a new VEE any better?
  - After all, it is not likely to be anywhere
- If extending the JVM takes too long, why is it better to create a new VEE?
  - Surely, that will take even longer?

Some thoughts on how to do better

- Design the VEE to support dynamically typed languages.
  - Statically typed languages are just dynamically typed languages where all the type checks can be discharged by the compiler. That compiler may as well be the VEE JIT.
- Use late-binding and polymorphic data containers to avoid favoring a particular type system.
  - For example, in the CLR a dynamically typed language can use objects created by statically typed languages, but the latter cannot use objects created by the dynamically typed languages (except via Reflection).

No VEE can be ubiquitous, but

- Host new VEE on both JVM and CLR.
  - On demand translation from new IL to JVM and CLR bytes codes might be feasible.
  - There will be a performance penalty, but perhaps not too much.
- Import JVM and CLR libraries by translating their byte codes to the new IL.
  - This means we that we don't have to start over and don't have to wait for broad based deployment and adoption before getting to critical mass.
  - However, supporting all the various kinds Native callouts may be tricky and labor intensive.
- Persuade the major web browsers to adopt the new VEE as an alternative serialization format for JavaScript (and its successors).

Basics

- Register based IL
  - For a large number of virtual registers
- Highly regular instruction set
  - Do not burden language front-ends with the peculiarities of IL compression.
- First class tagged values
  - Polymorphism must be supported by the instruction set and be understood by the GC

Interpreter vs. JIT

- There is still disagreement on the desirability of a VEE that generates machine code for all IL code.
- I am in the camp that wants to JIT everything.
- It would be interesting, however, to try and settle this question definitively, at least in the context of a new VEE.

In-line caches

- Crucial for high performance dynamic languages.
- Also very good for de-virtualizing virtual method calls in OO languages.
- Can also help with things such as transactional memory.
- Could be “free” if a tracing JIT is assumed.

Functions and closures

- Stack frames need to be accessible to language runtimes.
  - JavaScript runtimes need to be able to capture references to live stack frames, dynamically add locals to a stack frame, walk stacks and reflect over stack frames. This is very costly to do if not supported by the VEE.
- Tail-call recursion must be supported.
- Creating a VEE thread must be very cheap.

Vector instructions

- Not well supported on current generation of VEEs.
- The next few years will probably see support for vector instructions via libraries of intrinsic methods/functions understood by the JIT.
- I would prefer to see dedicated IL instructions because I'd like method calls to be dynamically bound, which is not a good match for intrinsics.

Many-core

- It is not clear that current dominant approaches to garbage collection will scale across hundreds of cores.
- Applications will have to become more aware of caches and the need to avoid invalidating cache lines that are shared by many cores. The VEE and its GC will have to provide ways for the applications to control their cache impact.
- New Thread libraries.
- Control over scheduling of threads.
- Message passing primitives (e.g. Barrelfish?)

Non uniform memory architectures

- If the VEE provides the illusion that all of memory is one big shared heap, then it is easy to ignore the existence of NUMA, but difficult to generate code that will exploit NUMA.
- Look into having multiple heaps. Perhaps support linear types and read-only objects as a means for enabling communication between heaps while keeping them disjoint.

Grids and Clouds

- We need better tooling and libraries that work well with grids and clouds.
- This creates an inflection point that makes thoughts of a new VEE more attractive.

Versioning and long-lived data formats

- Preserving data without the accompanying code makes little sense.
- Nor does preserving code without the accompanying VEE.
- Preserving code in the form of a nominal type system (e.g. as JVM class files) creates versioning problems.