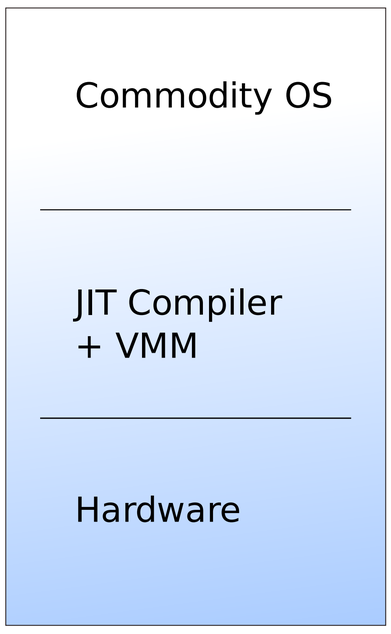
JOVE: A Framework for JIT Compilation of an Operating System in a Virtualized Environment

Will Dietz, Andrew Lenharth, Vikram Adve
University of Illinois at Urbana-Champaign

What is JOVE?

- JIT Compilation of a Commodity OS in a Virtualized Environment
- Extension of Secure Virtual Architecture (SVA) work from LLVM Researchers
- Foundation for new work



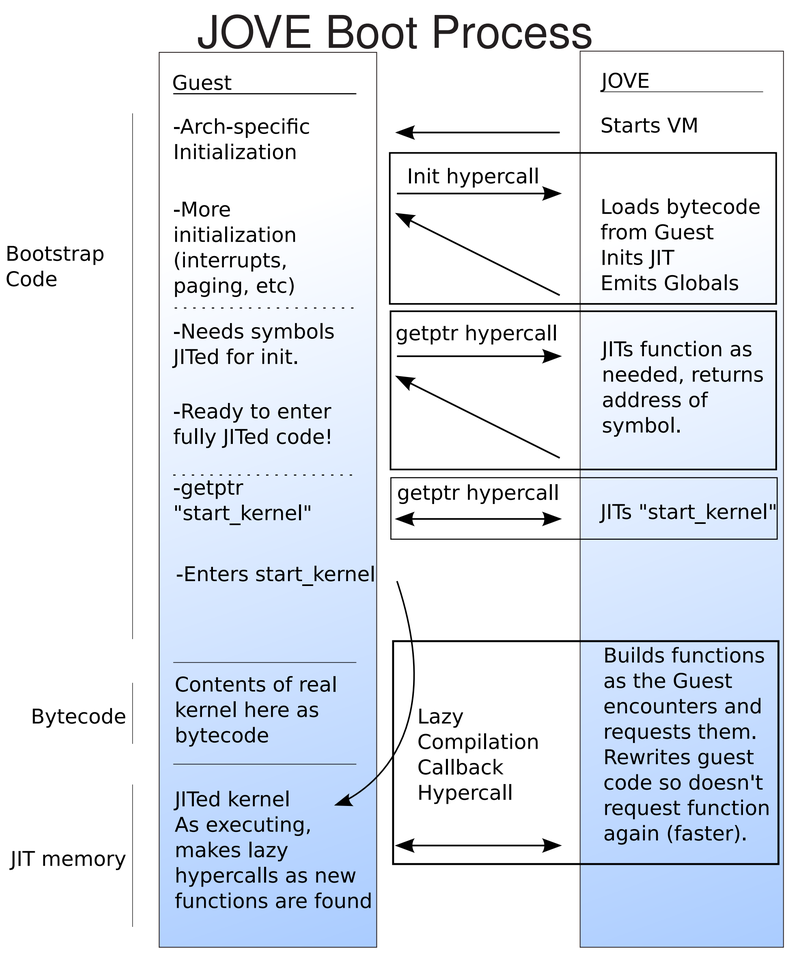
Why is this important?

- Improvements to SVA
  - Provides Trusted Boot mechanism to SVA
    - Can boot arbitrary bytecode kernels and apply safety to them
  - Kernel module loading safety
- Bring JIT benefits that userspace enjoys to the kernel
  - Foundation for new applications with exciting potential
    - Performance
    - Security
    - Portability
- Provides a single general mechanism, namely a JIT for
  - Instrumentation
  - Profiling
  - Patching

How is this done today?

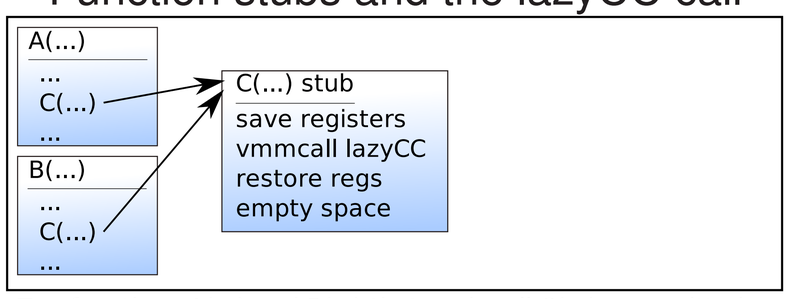
It's not

System Description

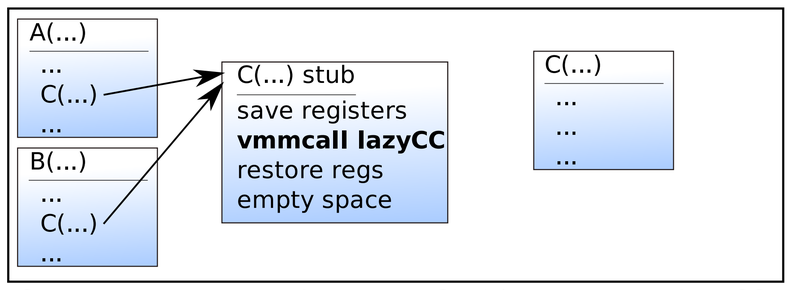


- Uses Hypercalls to interact with Guest
  - `init`
  - `getptr`
  - `lazyCC`
- Only modifies the bootstrap code
- Compiles the kernel as functions and globals are needed
- Hardware virtualization (AMD-V, Intel-VT)

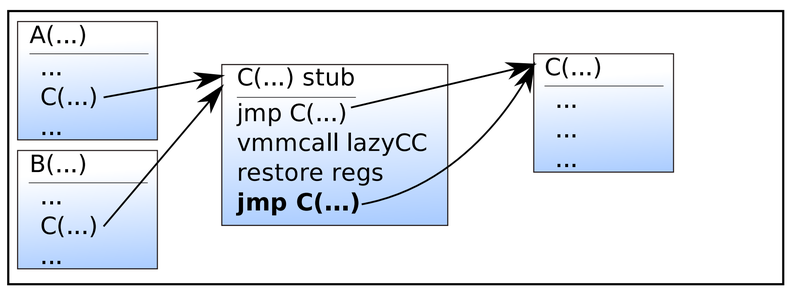
Function stubs and the lazyCC call



Two functions A(...) and B(...) that each call C(...) are emitted. Function stub for C(...) is emitted to handle their call to C(...).



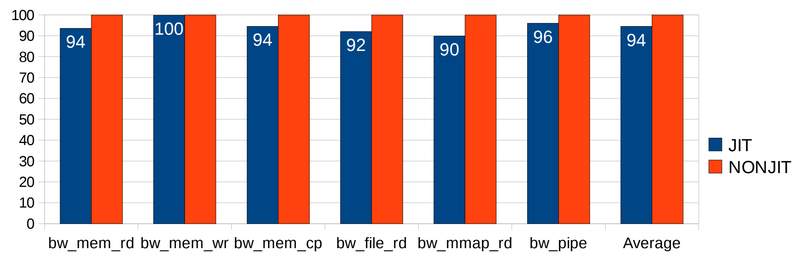
Code execution enters A, then the C(...) function stub. LazyCC hypercall is issued and JOVE emits C(...).



JOVE patches the function stub so future use is immediately redirected. Code execution returns from hypercall, registers are restored and executes the jmp to C(...).

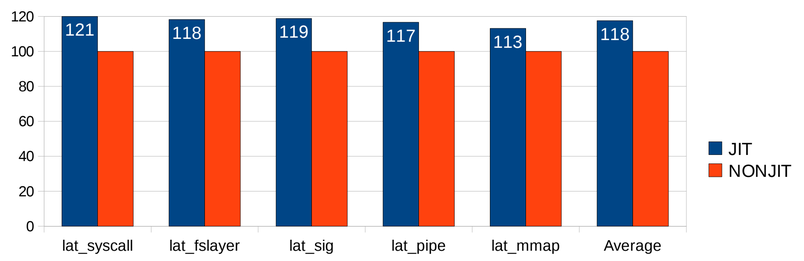
Performance

94% Bandwidth



HBench-OS Results -- Bandwidth -- Percentage of NONJIT

17.5% Latency Overhead



HBench-OS Results -- Latency -- Percentage of NONJIT

- Strong performance numbers
- Completeness of system
- Future work has solid foundation to build on

Future Work

- Dynamic Optimizations
  - Online profiling
    - Debugging/tracing applications
    - Dynamic optimizations based on profile feedback
  - Code specialization
    - Take advantage of run-time information
    - Especially useful in e.g. drivers
  - Architecture-specific optimizations
    - Take advantage of hardware features that weren't known at build time
  - Replace Interpreters within the kernel
    - e.g. Packet filter interpreter could be replaced with JIT calls to emit native code
- Security Applications
  - Online patching security vulnerabilities
    - No need to reboot
    - Full source not needed, can replace an entire function using only compiled replacement
- Portability
  - Bytecode portability
    - Because code is generated at run-time, there is the potential to have a cross-platform kernel
  - Cross-platform drivers
    - Given stable ABI a single driver binary could work on many platforms

University of Illinois at Urbana-Champaign Mail: {wdietz2,alenhar2,vadve}@uiuc.edu WWW: http://llvm.cs.uiuc.edu