Cross-ISA Debugging in Meta-circular VMs

VMIL'17, 24 Oct 2017

Christos Kotselidis Andy Nisbet Foivos S. Zakkak Nikos Foutris

MANCHESTER 1824
The University of Manchester

Meta-circular VMs

- VMs written in the language they are meant to implement
- Meta-circular VMs use their own compilers to build themselves (create a *boot image*)

Development challenges

Porting Meta-circular VMs to new Instruction Set Architectures

- No existing compilers for the target ISA
- Need to implement at least all the components required to build the boot image
- Compiler bugs prevent the VM from starting
- Testing of the compiler cannot be performed without the VM running
- No sufficient tools (e.g. testing framework) to assist in the above process

Debugging the boot image

- Exceptions might appear before we can even print and/or trace them

Development challenges

Porting Meta-circular VMs to new Instruction Set Architectures

- No existing compilers for the target ISA
- Need to implement at least all the components required to build the boot image
- Compiler bugs prevent the VM from starting
- Testing of the compiler cannot be performed without the VM running
- No sufficient tools (e.g. testing framework) to assist in the above process

Debugging the boot image

- Exceptions might appear before we can even print and/or trace them

Our approach

Porting Meta-circular VMs to new Instruction Set Architectures

Cross-compile unit tests and run using virtualization

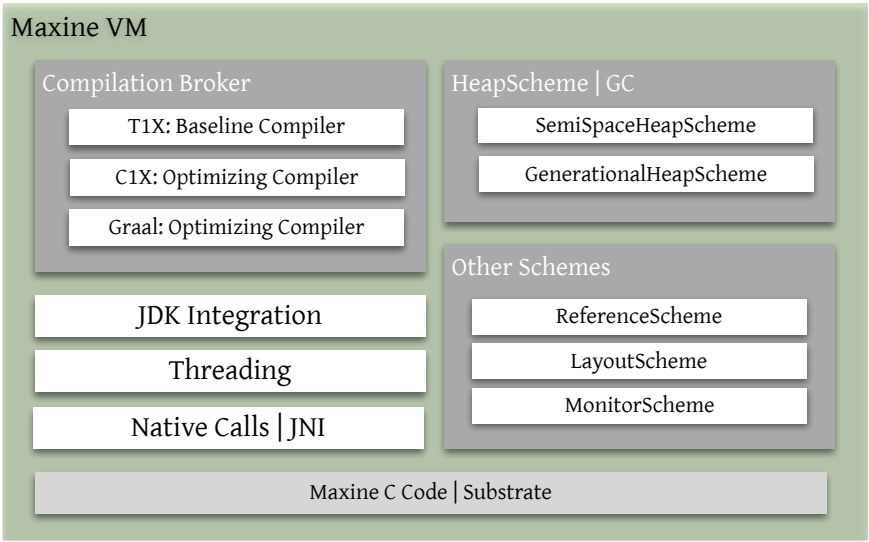
Debugging the boot image

Injection of special assembly instructions acting as markers that helps us map the failing native function (even if inlined) to the corresponding VM method

Maxine VM

- Meta-circular VM
- Originally a Sun and Oracle Labs project,
  maintained by the University of Manchester since project kenai shut down
  `https://github.com/beehive-lab/Maxine-VM`
- Ported to ARMv7 using the tools presented in this talk
- Ongoing porting to ARMv8 using the same tools

Maxine VM Outline



Maxine's Compilers

T1X Template based compiler, used instead of an interpreter

- \+ Fast compilation
- – Minimal to no optimizations

C1X Optimizing compiler (C1 ported to Java)

- \+ Optimizing
- – Slower Compilation

Graal Optimizing compiler (Alternative or complementary to C1X)

- \+ Aggressive Optimizations
- – Slower Compilation
- – Experimental

QEMU-based Cross-ISA Debugging Toolchain

- Initialization (Create a code buffer, set expected values, etc.)
- Code generation (fill the buffer with the generated code)
- Creation of an executable (assemble, compile, and link)

```
$ arm-unknown-eabi-gcc -c -march=armv7-a -g test_armv7.c -o test_armv7.o
$ arm-unknown-eabi-as -mcpu=cortex-a9 -g startup_armv7.s -o startup_armv7.o
$ arm-unknown-eabi-ld -T test_armv7.ld test_armv7.o startup_armv7.o -o test.elf
$ arm-unknown-eabi-objcopy -O binary test.elf test_armv7.bin
```

- Run binary using QEMU and gdb

```
$ qemu-system-arm -cpu cortex-a9 -M versatilepb -m 128M -nographic -s -S -kernel
    test_armv7.bin
$ arm-none-eabi-gdb # target remote localhost:1234
```

- Validate results (compare register values to expected ones)

Supported Kind of tests

- Individual Assembly Instructions
- T1X Compiled Methods
- C1X Compiled Methods

Benefits

- Ease porting to new ISAs by enabling cross-ISA debugging
- Speedup regression and enhance productivity by:
  - testing cross-ISA compilers on more powerful than the target machines (e.g. x86 vs ARMv7)
  - not building the image for unit testing

```
1  $ time mx image
2    132.67s user 3.60s system 421% cpu 32.365 total
```

Examples

Live demo at the end (if time permits)!

Unit Test for ARMv7 add Assembly Instruction

```
1  public void test_add() throws Exception {
2    for (int i = 0; i < 10; i++) {
3      asm.movImm32(ConditionFlag.Always, ARMV7.cpuRegisters[i], expectedValues[i]);
4      asm.addq(ARMV7.cpuRegisters[i], expectedValues[i]);
5
6      expectedValues[i] += expectedValues[i];
7      testValues[i] = true;
8    }
9
10   generateAndTest(expectedValues, testValues, bitmasks, asm.codeBuffer);
11 }
```

Unit Test for ARMv7 T1X add Template

```
1  public void test_add() throws Exception {
2    initializeFrameForCompilation();
3
4    t1xCompiler.do_iconst(1);
5    t1xCompiler.do_iconst(2);
6    t1xCompiler.do_iadd();
7
8    expected[0] = 3;
9
10   generateAndTest(expectedValues);
11 }
```

BC_iadd2 Unit Test

```
1  package jtt.bytecode;
2
3  /*
4   * @Harness: java
5   * @Runs: (1b,2b)=3; (0b,-1b)=-1; (33b,67b)=100; (1b, -1b)=0;
6   * @Runs: (-128b,1b)=-127; (127b,1b)=128;
7   */
8  public class BC_iadd2 {
9
10   public static int test(byte a, byte b) {
11     return a + b;
12   }
13
14 }
```

Unit Test for ARMv7 C1X Compilation

```
1  public void test_C1X_jtt_BC_iadd2() throws Exception {
2    byte[] argsOne = {1, 0, 33, 1, -128, 127};
3    byte[] argsTwo = {2, -1, 67, -1, 1, 1};
4    initTests();
5    String klassName = getKlassName("jtt.bytecode.BC_iadd");
6    List<TargetMethod> methods = Compiler.compile(new String[] {klassName}, "C1X");
7    initializeCodeBuffers(methods, "BC_iadd2.java", "int test(byte, byte)");
8
9    for (int i = 0; i < argsOne.length; i++) {
10     int expectedValue = jtt.bytecode.BC_iadd2.test(argsOne[i], argsTwo[i]);
11     String functionPrototype =
12       ARMCodeWriter.preAmble("int", "int, int ", Integer.toString(argsOne[i]) + ","
             + Integer.toString(argsTwo[i]));
13     Object[] registerValues = generateObjectsAndTestStubs(functionPrototype,
           entryPoint, codeBytes);
14     assert (Integer) registerValues[0] == expectedValue;
15   }
16 }
```

Limitations

We still can't test the following without booting the VM

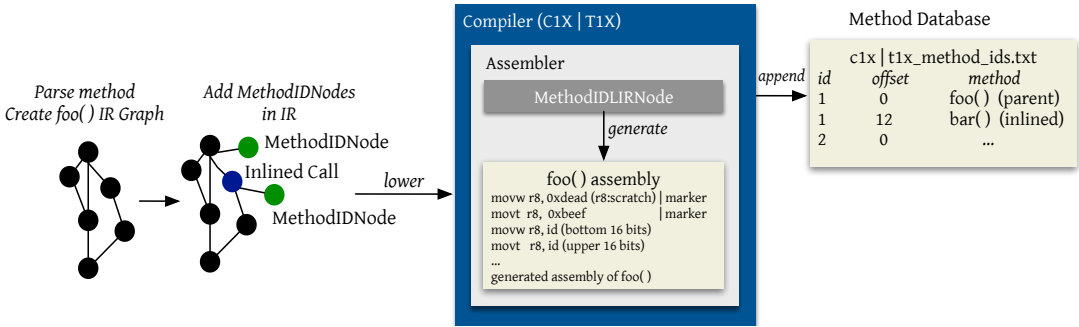
- Object allocation
- Garbage collection
- Synchronization
- Inline assembly (compiler stubs, adapters from T1X to C1X, etc.)

As a result, starting the VM on the target ISA is expected to result in a number of hard faults (e.g. `SIGSEGV`)

Tracing Runtime Faults

- Extend C1X and T1X to inject special assembly instructions:
  1. that act as markers that we can easily find when going through a `gdb` backtrace
  2. holding a unique, per method, ID
- Create a file, mapping the unique IDs to the corresponding Java methods
- On failure:
  1. Run the VM inside gdb
  2. Print the backtrace
  3. Look for the closest marker before the failing instruction
  4. Obtain the method ID and look it up in the map file

Tracing Runtime Faults Outline



Demo Time

Conclusions

- Enable Cross-ISA debugging
- Speedup development
  - Unit testing compilers without building the image
  - Unit testing on stronger machines than the potentially weaker target
- Improve bootstrap debugging by injecting special instructions

Thank You!

Cross-ISA Debugging in Meta-circular VMs

VMIL'17, 24 Oct 2017

Christos Kotselidis Andy Nisbet Foivos S. Zakkak Nikos Foutris

MANCHESTER 1824
The University of Manchester