Cross-ISA Debugging in Meta-circular VMs

Christos Kotselidis
The University of Manchester
Manchester, United Kingdom
christos.kotselidis@manchester.ac.uk

Andy Nisbet
The University of Manchester
Manchester, United Kingdom
andy.nisbet@manchester.ac.uk

Foivos S. Zakkak
The University of Manchester
Manchester, United Kingdom
foivos.zakkak@manchester.ac.uk

Nikos Foutris
The University of Manchester
Manchester, United Kingdom
nikos.foutris@manchester.ac.uk

Abstract

Extending current Virtual Machine implementations to new Instruction Set Architectures entails a significant programming and debugging effort. Meta-circular VMs add another level of complexity towards this aim since they have to compile themselves with the same compiler that is being extended. Therefore, having low-level debugging tools is of vital importance in decreasing development time and bugs introduced.

In this paper we describe our experiences in extending Maxine VM to the ARMv7 architecture. During that process, we developed a QEMU-based toolchain which enables us to debug a wide range of VM features in an automated way. The presented toolchain has been integrated with the JUNIT testing framework of Maxine VM and is capable of executing from simple assembly instructions to fully JIT compiled code. Furthermore, it is fully open-sourced and can be adapted to any other VMs seamlessly. Finally, we describe a compiler-assisted methodology that helps us identify, at runtime, faulty methods that generate no stack traces, in an automatic and fast manner.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Software testing and debugging**;

Keywords just-in-time compilation, debug, cross-isa, qemu, maxine vm, metacircular

ACM Reference Format:
Christos Kotselidis, Andy Nisbet, Foivos S. Zakkak, and Nikos Foutris. 2017. Cross-ISA Debugging in Meta-circular VMs. In *Proceedings of ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3141871.3141872

VMIL'17, October 24, 2017, Vancouver, Canada

1 Introduction

Collectively, *managed programming languages* implemented on top of *managed runtime environments* have a large share in today's generated source code. Java, in particular, since its original design has paved into almost every aspect and domain of computing. Spanning from desktop, to server and Big Data applications Java and consequently the Java Virtual Machine (JVM) [8] has been widely used over the years resulting in a vast and healthy source code base and library APIs. Other than Java, a number of other programming languages such as Scala [11], Ruby [9], and Truffle-based DSLs [20], are also built atop the JVM taking advantage of its advanced Just-In-Time (JIT) compilation infrastructure, automatic memory management in the form of Garbage Collection (GC), and other features that enable highly productive programming.

The main driving force of managed programming languages is architecture portability adhering to the "write-once-run-everywhere" dogma. For example, Java uses a two-tiered compilation process in which the source code is first compiled to a portable bytecode format before it can be interpreted or JIT compiled by the interpreter and the compilers of the JVM. In order to execute a program across machines with different instruction set architectures (ISAs), programmers do not have to explicitly recompile or port their code. On the contrary, the Virtual Machine (VM) engineers provide cross-ISA interoperability by porting the VMs across different ISAs.

The process of extending a VM to run on a new ISA is a tedious and time consuming task because of the amount of work entailed. Modern VMs, usually integrate a number of compilers, interpreters, and in-line assembly code that all must work in harmony to achieve correct execution. Porting a VM to a new ISA typically requires the development and testing of numerous compiler back-ends for assembly generation, as well as additional parts that VMs typically utilize. Typically such VM parts are compiler intrinsics, compiler stubs, adapters for transitions between code compiled by different compilers, and others.

In addition to production-quality JVMs, such as Oracle's HotSpot [10] and IBM's J9 [5], a number of research VMs such as the Maxine VM [18] and Jikes RVM [1] also exist.

A number of these research VMs, mainly designed for the Java language, differ from the production-quality ones due to their "meta-circular" nature. Instead of being developed in statically compiled languages such as C or C++ they are implemented in the programming language they are designed to execute; in this case Java. This meta-circular nature allows for easier and faster prototyping of new language features, since Java is a higher level language than C and C++. Additionally it allows for experimentation with alternative mechanisms on the core of the VM, making meta-circular VMs suitable for research. However, this meta-circular nature adds another level of complexity when porting the VMs to a new ISA because they are compiled by the same compilers that need to be ported to that new ISA. Consequently, VM engineers face difficulties in testing cross-ISA implementations because any bug introduced during the compiler implementation will prohibit the VM to execute correctly and thus test the compiler that is being ported!

In this paper, we describe our experiences in porting the Maxine VM from the 64-bit x86 to the 32-bit ARMv7 architecture. In addition to transitioning Maxine VM from 64 to 32 bit execution, we also introduced a new ISA to it. The aforementioned meta-circular-derived problems of testing and validating the new compiler back-ends led us to the development of a complete QEMU-based toolchain that allows us to perform a high-coverage external debugging and low-level assembly analysis. The presented toolchain has been integrated in the JUNIT testing framework of Maxine VM and is capable of executing from simple assembly instructions to fully JIT compiled code. Furthermore, it is fully open-sourced as of Maxine VM v2.1[1] [6] and can be adapted to any other VMs seamlessly.

In summary, in this work we make the following contributions:

- We describe our experiences in porting Maxine VM, the state-of-the-art research VM, from the 64-bit x86 to the 32-bit ARMv7 architecture.
- We introduce a complete QEMU-based toolchain that enables high-coverage external debugging and low-level assembly analysis bypassing the problem of "meta-circularity" when introducing a new ISA.
- We describe a compiler-assisted methodology for identifying, at runtime, faulty methods that generate no stack traces, in an automatic and fast manner.
- We provide guidelines of how to use the tool and techniques introduced in tandem with existing tools to perform end-to-end execution and debugging of selected methods and/or as part of a JUNIT framework.

The rest of the paper is organized as follows: Section 2 shortly describes Maxine VM and its 64 to 32 bit transition.

[1]https://github.com/beehive-lab/Maxine-VM

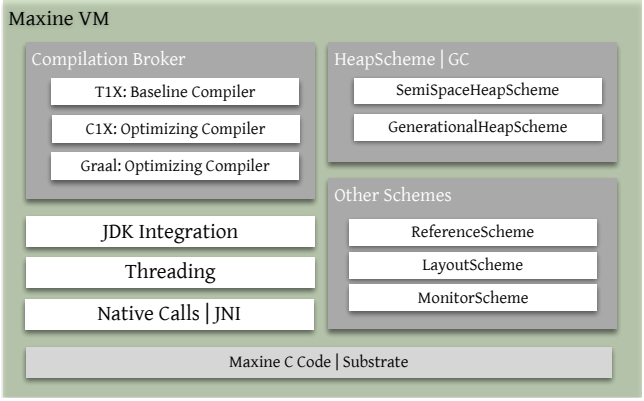

Figure 1. Maxine outline.

Section 3 describes the introduced toolchain and methodologies along with a number of developer guidelines. Finally, Section 4 concludes the paper.

2 Background

In this work, we focus on Maxine VM and therefore, from hereafter, we use Maxine VM terminology which we introduce in this section.

2.1 Maxine VM

Maxine VM [6, 18] is a meta-circular VM, for Java, written in Java. Initially developed by Sun and Oracle Labs, it is now maintained as an open source project and has spawned a number of side projects [4, 7, 14].

Maxine VM follows a modular approach in which the components that comprise the VM, called *schemes*, can be defined during build time. Examples of such schemes, are the *layout scheme* defining the object layouts, the *heap scheme* defining the garbage collection algorithms, and the *reference scheme* defining the type of object referencing. Maxine VM's variety of schemes facilitates the experimentation and prototyping of various features. Figure 1 gives an overview of the modular architecture of Maxine VM and its schemes.

Currently, Maxine VM offers three compilers:

1. **T1X:** A fast template compiler similar to an interpreter. T1X is essentially a fast template JIT compiler that generates less optimal code than other more aggressive optimizing JIT compilers but has low compilation times; essentially, serving the role of an interpreter found in other VM implementations.
2. **C1X:** A JIT optimizing compiler that compiles frequently executed code based on the statistics gathered by T1X.
3. **Graal:** An aggressive JIT optimizing compiler, originated by C1X, that has formed its own project within Oracle Labs [12]. As of Maxine v2.0, Graal has been

integrated to Maxine VM and can be used as a replacement or in collaboration with C1X.

The selection of the compiler follows the same logic with scheme selection. During build time, we choose which optimizing compiler to use (T1X is always enabled by default).

In order to build the Maxine VM, a host Java Development Kit (JDK) is used to compile its source code into bytecode. Consequently, the boot image of Maxine VM is generated with the help of a host JVM. The boot image is an ahead-of-time compiled binary which contains the necessary code for booting and initializing the VM. Several sub-components of Maxine VM such as the locking machinery and GC code reside inside the bootimage. Upon boostraping Maxine VM, the bootimage is loaded into memory updating all inter-bootimage references and through a thin layer of C code, called the substrate, execution starts.

A critical point of this building process is the fact that the bootimage generated code is compiled by one of Maxine VM's optimizing compilers. That means that in order to build and execute the Maxine VM, we must ensure that its compilers, or at least the ones we choose, produce valid code; otherwise the boot image code will fail to execute correctly.

The problem of meta-circularity The fact that Maxine VM's bootimage is ahead-of-time compiled by one of its runtime compilers, creates a paradox when trying to port it to a different ISA. Ideally, in order to test the validity of the generated code for the new ISA, we would like to run the VM and through its unit testing framework to start testing individual compiled methods under the new ISA. However, this is not possible in our case since we can not boot the VM if we do not ensure that the compilers have been ported to the new ISA. At the same time, we can not test the compilers through the VM's testing framework if we can not boot the VM. This problem led us to develop an external QEMU-based toolchain infrastructure that enables us to circumvent the bootimage generation and test a wide range of the VM's features offline. We describe this toolchain along with additional debugging infrastructure in Section 3.

In theory, we could use another VM to compile ahead-of-time the bootimage but this would "break" the meta-circular nature of Maxine VM. Furthermore, additional changes to the build toolchain of Maxine VM would be required.

2.2 64 to 32 bit Transition

Apart from the challenges of porting Maxine VM to the new ARM ISA, we also encountered a number of challenges during the transition from 64 to 32 bit. We discuss below the two most significant ones.

2.2.1 Object Identification

Typically, Java objects have an associated hashcode which uniquely identifies them. Hashcodes are 32 bit long and stored in objects' headers for fast retrieval. Depending on the VM implementation, objects have additional fields added in their data layout that assist in various VM-related functions, such as synchronization and GC. Furthermore, every VM implementation has a different number or size of fields in the object headers.

Maxine VM, uses two additional words for plain objects and three for arrays. Regarding objects, the two extra words store the class pointer, the object's hashcode, and some other metadata used for locking, threading, etc. Arrays follow a similar approach with plain objects with the addition of an extra word that stores their size.

In a 64 bit machine, an object's hashcode can be stored in the second word of their header leaving another 32 bits free for extra metadata storing. Transitioning to a 32 bit machine however, leaves no space for these extra metadata bits since the hashcode occupies the entire word.

A potential solution would be to truncate the 32 bit hashcode to less bits by providing a custom implementation of the hashcode function. Although this solution would increase aliasing problems, it would not require any additional changes to the objects' layouts.

However, this is not possible in the context of a meta-circular VM, like Maxine VM, since some objects are being created and stored during boot image generation by the host JVM. The host JVM being unaware of the custom hashcode implementation of Maxine VM, will store the full 32 bit values in the bootimage leading to miss-identification when those objects are requested by Maxine VM during runtime. Since augmenting the host JVM to also use truncated hash codes is not a vital solution, we opted for adding an additional word in objects' headers on 32 bit architectures. This leads to a 10% extra heap space overhead as showcased by [15]. Reducing the additional space required by object headers has been previously studied in [2, 16].

2.2.2 Register Allocation

Maxine VM uses a version of the linear register allocation described in [13]. In fact, it is a Java version of the register allocator found in the HotSpot JVM [17]. Since Maxine VM was designed primarily for 64 bit architectures, all the extensions that would enable efficient register allocation on 32 bit machines were absent. Thus, we implemented the missing 32 bit register support based on the SPARC version of the register allocator[2].The added functionality allows the allocation of adjacent registers for long and double values, as well as the allocation of double registers starting always from odd numbering, as specifically required by the ARM architecture.

3 Toolset and Methodologies

In this Section we describe the QEMU-based [3] toolchain we developed that enables us to debug a wide number of

[2]http://hg.openjdk.java.net/jdk8/jdk8/hotspot

features when porting the meta-circular Maxine VM to a new ISA. Furthermore, we discuss how someone could combine the introduced toolchain with existing tools as well as additional debugging functionality we added in Maxine VM, to enable end-to-end bug tracking and debugging during runtime (from source code down to assembly).

3.1 QEMU-based Toolchain

Figure 2 depicts the outline of our toolchain.

As shown in Figure 2, the existing unit testing framework of Maxine VM communicates with the `MaxineTester` class which is responsible for the whole coordination. In order to run a unit test the following steps are performed in order:

1. **`MaxineTester` initialization:**
   When the unit tests are invoked, they initialize (1) the `MaxineTester` which resets all internal state and QEMU output files. This process is performed per unit-test by placing the `initialize` call inside every unit test. This is due to instabilities we observed during QEMU execution which led to the unit testing framework *hanging* during nightly regressions. Hence, we followed a more conservative approach in which the QEMU is re-initialized during every unit test execution. After the initialization completes a code buffer is returned to the unit test (2) which serves as the placeholder for the generated assembly code.
2. **Generating the code of the unit test:**
   Depending on the nature of the unit test; i.e., simple assembly instruction, T1X or C1X testing, the code buffer is filled in a different manner. We describe those differences in Section 3.2. When the code buffer is filled, it is passed to the `MaxineTester` for QEMU emulation (3).
3. **Composing the binary for QEMU emulation:**
   After receiving the code to be tested, the `MaxineTester` assembles the binary that will be passed to QEMU for emulation (4). The process of creating the binary file (`test.bin`) is as follows:
   Initially, we generate the assembly code of two helping assembly files (`asm_startup.s` and `entry.s`) which are linked together with the binary code of the unit test. Consequently, the code buffer that contains the assembly code of the unit test is inlined to a C file (`codebuffer.c`) and a function pointer to its first entry is installed. Finally, the actual test (`test.c`) that links together the `codebuffer.c` and the two assembly files (`asm_startup.s` and `entry.s`) is compiled and the `test.bin` binary is formed. The generated `test.bin` binary, in essence, contains code that helps running the binary code injected through the code buffer (i.e. a main function with a function pointer invocation to the embedded code of the code buffer).
4. **Performing the QEMU emulation:**
   The next step, after creating the `test.bin` binary file that contains our unit test, is the QEMU emulation (5). In our case, since we port Maxine VM to the ARM ISA, we simulate a Cortex-A15 processor, as shown in Figure 2. QEMU will run the binary emulating the ARM core we defined, and upon completion, it will dump the register file to an output file defined in the `MaxineTester` class.
5. **Validating the execution output:**
   The output of QEMU executing a unit test is the dump of the register file of the emulated processor. The output register file is validated against the expected values as set in the unit test definition. Depending on the nature of the unit test, such definitions might be explicit (e.g. in which register we expect a certain value to be written) or implicit (e.g. the return value of a C1X compiled method which is written in register `r0` according to the ARM calling convention).

The process described above is performed for every unit test implemented in the unit test framework. In the next Section we provide some examples regarding the unit test implementations.

3.2 Implementing a Unit Test

Our QEMU-based testing infrastructure can execute three kinds of unit tests that can exercise:

1. **Individual Assembly Instructions:**
   This kind of tests regard individual assembly instructions that mainly exercise the correct instruction encoding while adding a new ISA instruction to the assembler back-end.
2. **T1X Compiled Methods:**
   This kind of tests assess the validity of the code generated by the T1X compiler.
3. **C1X Compiled Methods:**
   Unit tests that exercise C1X compiled code are the most complicated ones since they necessitate a significant part of the compilation process to work offline (i.e. without runtime support).

Note that the described methodology has not been ported to the Graal compiler since the ARM port supports only the T1X and C1X compilers. In the following sections we present one unit test example for each kind of unit test described above.

3.2.1 Testing Individual Assembly Instructions

Listing 1 shows a simplified version of the unit test for the `add` instruction. As shown in the Listing, initially we reset the array with the expected values of the simulation. Then, we perform a number of `mov` instructions that move the values of the expected values array into the ARM registers (`r0-r9`) in ascending order. Consequently, we perform the additions of the contents of the registers with the values from the expected array, writing the results back to the registers.

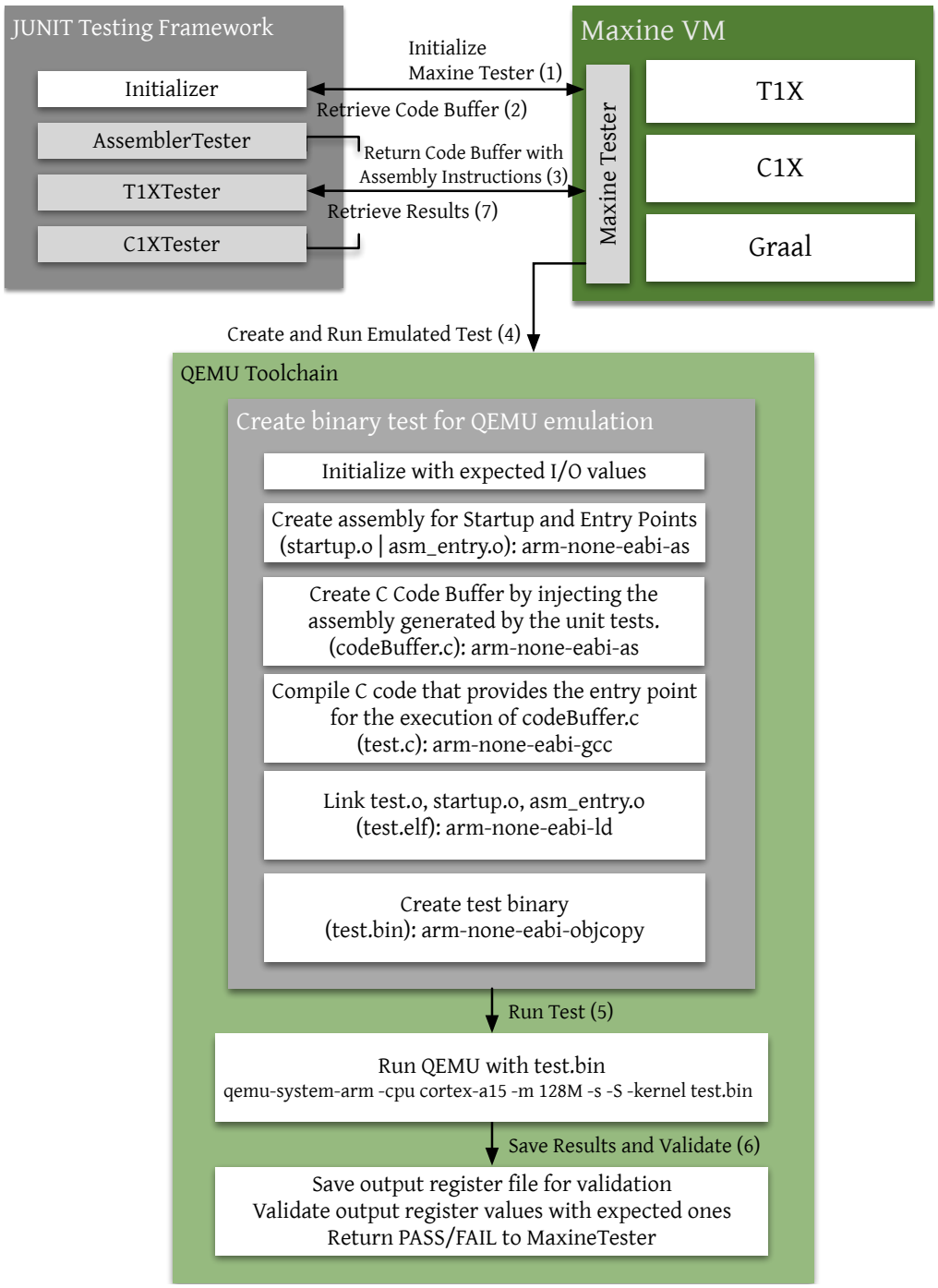

Figure 2. QEMU based toolchain outline.

Finally, we update the expected array with the values that the emulation should produce (r_i= `2 * expected[i]`). After we append all the instructions to the code buffer, we pass it along with the expected values array for generation, emulation, and validation as described in Section 3.

3.2.2 Testing T1X Compiled Methods

Listing 2 shows a simplified version of the unit test for T1X's template of the add instruction. As shown, this test differs from the simpler individual assembly instruction test. Firstly, we initialize a frame for the T1X template to be tested. The frame initialization will add in the code buffer all the necessary instructions that form the prologue of a T1X compiled

Listing 1. Unit test for ARM add assembly instruction.

```
private static int[] expected =
  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

public void test_add() throws Exception {
  initializeExpectedValues();
  asm.codeBuffer.reset();
  for (int i = 0; i < 10; i++) {
    asm.movImm32(ConditionFlag.Always,
        ARMV7.cpuRegisters[i], expected[i]);
    asm.add(ARMV7.cpuRegisters[i], expected[i]);
    expected[i] *= 2;
  }
  generateAndTest(expected, asm.codeBuffer);
}
```

Listing 2. Unit test for ARM T1X add template.

```
public void test_add() throws Exception {
  initializeFrameForCompilation();
  t1xCompiler.do_iconst(1);
  t1xCompiler.do_iconst(2);
  t1xCompiler.do_iadd();
  expected[0] = 3;

  int[] registers = generateAndTest(expectedValues);
  assert expected[0] == registers[0];
}
```

template. Consequently, we call twice the `do_iconst` T1X template which generates code for pushing onto the stack the values 1 and 2. Finally, we call the `do_iadd` T1X template which pops two values from the stack and performs the addition. At the end of the execution of the add template, register `r0` should have the result of the adding the operators (`r0=3`). Therefore, the assertion checks if `r0` from the returned register file is equal to the expected value of 3.

3.2.3 Testing C1X Compiled Methods

Listing 4 shows a simplified version of the unit test of the `jtt.bytecode.BC_iadd2` method of Listing 3. This test adds two byte operands and returns an integer result. Our objective is to compile that method with C1X and call it with all the combinations of the operand parameters defined in the test. Consequently, the test will be emulated on the ARM architecture and its results will be validated against the expected ones.

As shown in Listing 4, the first step is to initialize two byte arrays that hold the input operands of the unit test. Then, we perform a series of initialization stages which essentially create offline instances of our optimizing compiler (C1X). The next step is to invoke the C1X compiler to compile our method by passing a canonical representation of its class name.

Then, we call the function `initializeCodeBuffers` that initializes the code buffer sizes and assigns values to the following variables: *a)* `entryPoint` which holds the offset of the entry point of the compiled method, and *b)* `codeBytes` which holds the binary of the compiled method. After we generated the code of the compiled method, our next step is to compose the unit test and pass it to QEMU for emulation and validation.

Listing 3. BC_iadd2 unit test.

```
package jtt.bytecode;

/*
 * @Harness: java
 * @Runs: (1b,2b)=3; (0b,-1b)=-1; (33b,67b)=100; (1b,
     -1b)=0;
 * @Runs: (-128b,1b)=-127; (127b,1b)=128;
 */
public class BC_iadd2 {
  public static int test(byte a, byte b) {
    return a + b;
  }
}
```

To do that, we first need to create a C signature for the compiled method that corresponds to the Java one since the assembly code of the compiled method will be injected in the C test code as described in Section 3.

Finally, we pass to the `generateObjectsAndTestStubs` method all the input parameters. This method invokes the QEMU toolchain and runs the test. The returned values are asserted against the expected ones to test the validity of the execution. Again, by following ARM's calling convention since the result is an integer returned value, it should reside on register `r0`. Therefore, we check `registerValues[0]` that maps to `r0`.

Limitations: The QEMU-based toolset and methodology described in this section enables us to perform a large coverage of the newly implemented ISA instructions of Maxine. This partially, solves the problem of "meta-circularity" described in Section 2 since we do not have to achieve a functional bootimage in order to test the developed compilers. However, this technique does not provide full coverage. Various parts of the VM such as object allocation, garbage collections, and locking, that require runtime support can not be tested since they require a booted VM. Furthermore, inlined assembly code (i.e. not generated by Maxine's compilers) can not be tested. Such code typically relates to compiler stubs, adapters for transitioning between T1X and C1X code, etc.

In order to test and validate the above, we first need to obtain a working bootimage on the new ISA architecture. After achieving a wide coverage of the VM the next step is trying to boot it. At that stage, we encountered another problem with respect to "meta-circularity"; the lack of exception handling during VM bootstrap. The next section describes in detail this problem as well as the methodology we followed to track faulty methods at runtime, as well as during the VM bootstrap.

Listing 4. Unit test for ARM C1X compilation.

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

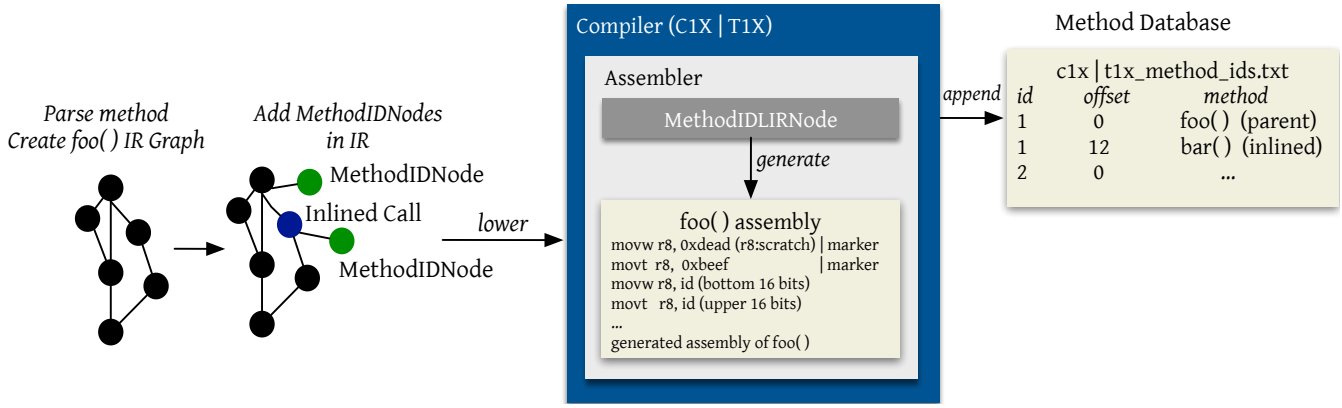

Figure 3. Adding MethodIDs to IR graph and generation of method database during compilation.

3.3 Tracking Faulty Methods at Runtime

When boostraping a meta-circular VM on a new ISA, many parts of the VM are still untested. Bootstraping the Maxine VM is a multi-stage process in which various parts of the VM are sequentially initialized. Since the exception handling mechanism of Maxine VM is initialized at later stages of the boot process, we encountered numerous hard `segfaults` that generated no exception stack traces. In order to identify such faulty methods during runtime, we revised the compiler-assisted methodology shown in Figure 3.

Since most of the times the VM was failing with a hard `SIGSEGV` violation, we were running it inside `gdb` in order to read the program counter (`PC`) and retrieve the address of the faulty instruction. Our next objective was to be able to map that address to the faulty compiled method in an automated manner. For that reason, we augmented Maxine VM's compilers (both T1X and C1X) to inject, during compilation, a unique identifier called `MethodID`.

The `MethodIDs` always start from a predefined number and are incremented in a thread-safe manner, using compare-and-swap instructions, during a method's compilation. In the generated code, the MethodID manifests as a pair of `movw`, `movt` instructions to the scratch register following another pair of `movw`, `movt` instructions that serve as a signature that we can later search for. At the same time, the compiler records that unique `MethodID` along with the method signature in a text file, essentially creating a database that maps `MethodIDs` to method signatures.

Upon a crash, we display the instruction sequence backwards inside `gdb` and search for the `MethodID` signature. Then, we inquire the `MethodID` inside the generated text file and detect the name of the faulty method. In combination with the tools described in Section 3.4 this methodology enabled us to efficiently debug the faulty code.

Another approach to discover the faulty methods would be to check the address of the faulty instructions, through the `PC`, against the code cache in order to find the boundaries of the faulty method. Although this works for the general case, it fails to detect the inlining boundaries. If a fault takes place inside a method inlined inside another method, it is crucial to be able to find the inner faulty method in order to narrow down the debugging effort. Our technique enables that by attaching a dedicated node to the IR graph of the compiled method. During the graph building phase of the compilation process, and during inlining, a dedicated `MethodIDNode` is attached to the IR graph before the inlined code.

The MethodIDNode consequently gets lowered, similarly to any other node, until its assembly gets emitted. The emitted assembly appends the `movw`, `movt` pair of the `MethodID` inside the compiled method. At the same time, the compiled method database also maintains, per compiled method, the offsets of the inlined entry points from the beginning of the parent method. This way, we can approximately discover any faulty inlined methods inside a parent method during runtime.

Limitations: The methodology described in this section, works for the general case with the exception of the inlining functionality in the context of highly aggressive optimizing compilers such as Graal. In that case, the faulty instructions can potentially be hoisted over the inlined method's marker and therefore mislead the developer regarding where the error occurred. Although this scenario can occur in highly optimized code, we have not encountered any occurrence in the context of the C1X compiler.

3.3.1 Maxine Inspector

Maxine VM comes with out-of-the-box debugging functionality provided by the Maxine Inspector [18]. Maxine Inspector is a powerful tool that allows its users to inspect a program's execution in real-time. By executing a program through the Maxine Inspector, one can monitor different parts of the VM including the stack(s), heap, registers, etc. Furthermore, one can step through or over executed instructions while being able to map assembly instructions back to the source code. Theoretically, a tool like Maxine Inspector makes the technique described in this section obsolete. Unfortunately, this does not hold true for the following three reasons:

1. The Inspector can not trace execution for a large number of instructions due to instability issues.
2. It has a significant performance penalty slowing down execution.
3. We need to provide a disassembler for the developed ARM architecture since it only supports x86.

For the aforementioned reasons, we developed the techniques described in this section. We hope to solve those issues in the future and constitute the Inspector as our tool for development and debugging.

3.4 Toolset Ecosystem

So far we explained two complementary approaches that help developers debug new ISA compiler back-ends and extensions. Although the original motivation for developing those tools originated by the "meta-circular" nature of Maxine VM, and the challenges that these kind of VMs have, they can be applied to any other VM implementation.

While the first tool regards offline testing of different kinds of generated code, the second methodology targets hard errors originated during runtime. By providing an automated way to find the origin of the faulty instructions, we can combine existing tools to track the compilation of a method from source code down to assembly.

In our experience, the IdealGraphVizualizer [19] and the c1Visualizer[3] tools proved valuable in finding the source of bugs. While the first tool visualizes the compilation process from the graph generation phase to the emission of the low-level IR (LIR), the second tool visualizes the code from LIR down to the generated assembly. Unfortunately, since we did not provide an ARM disassembler to c1Vizualizer we could not output the generated assembly. Luckily, by using the second technique described in this paper we could overcome this issue. By identifying the faulty method, we could compile it offline and use existing tools to reason about the source of bugs.

4 Conclusions

In this paper we presented two techniques that assist in debugging Maxine VM while porting it from x86 to the ARM ISA. The main motivation of the developed tools and techniques was problems related to the meta-circular nature of Maxine VM. The fact that Maxine VM is compiled by its own compilers results in the inability to use it as a testing vehicle for the developed compilers. To solve that issue, we implemented two complementary techniques that, although not providing full coverage, assist us in identifying and solving any bugs introduced when porting Maxine VM to a new ISA.

The first technique enables us to perform offline testing of generated code by using a QEMU-based toolchain. By running the generated code in emulation mode, we are able to debug from simple assembly instructions to methods compiled by the T1X and C1X compilers. The full automated toolchain has been integrated with Maxine's existing JUNIT functionality providing a wide coverage of features. The second technique enables the fast identification of faulty methods at runtime after a hard error. By using the compiler-assisted identification technique, we automate the process of finding the method signatures causing the hard faults that produce no stack trace inside `gdb`.

The tools and techniques presented in this paper have been tested on both ARMv7 and AArch64 ISAs and can be combined with existing tools providing an end-to-end debugging infrastructure. They are also available publicly (https://github.com/beehive-lab/Maxine-VM) under the GPL license as of Maxine VM's release v2.1 and can be adapted to other VM implementations.

Acknowledgments

This work is partially supported by the EPSRC grant PAMELA EP/K008730/1, and the EU Horizon 2020 grant ACTiCLOUD 732366.

[3] https://www.openhub.net/p/c1visualizer

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The JalapeñO Virtual Machine. *IBM Syst. J.* 39, 1 (Jan. 2000), 211–238. https://doi.org/10.1147/sj.391.0211

[2] David F. Bacon, Stephen J. Fink, and David Grove. 2002. *Space- and Time-Efficient Implementation of the Java Object Model.* Springer Berlin Heidelberg, Berlin, Heidelberg, 111–132. https://doi.org/10.1007/3-540-47993-7_5

[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[4] Georgios Goumas, Konstantinos Nikas, Ewnetu Bayuh Lakew, Christos Kotselidis, Andrew Attwood, Erik Elmroth, Michail Flouris, Nikos Foutris, John Goodacre, Davide Grohmann, et al. 2017. ACTiCLOUD: Enabling the Next Generation of Cloud Applications. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS '17)*. IEEE, 1836–1845.

[5] IBM. 2017. IBM J9. https://www.ibm.com/developerworks/java/jdk/. (2017).

[6] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[7] Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, et al. 2015. Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research. *In 9th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG)* (2015). http://arxiv.org/abs/1509.04085

[8] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley.

[9] Yukio Matsumoto and K Ishituka. 2002. Ruby programming language. (2002).

[10] Steve Meloan. 1999. *The Java HotSpot performance engine: An in-depth look.* Technical Report.

[11] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language.* Technical Report.

[12] Oracle. 2017. Graal Compiler. http://openjdk.java.net/projects/graal/. (2017).

[13] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sept. 1999), 895–913. https://doi.org/10.1145/330249.330250

[14] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. 2017. MaxSim: A Simulation Platform for Managed Applications. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '17)*.

[15] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. 2017. Type Information Elimination from Objects on Architectures with Tagged Pointers Support. *IEEE Trans. Comput.* PP, 99 (2017), 1–1. https://doi.org/10.1109/TC.2017.2709739

[16] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. 2007. Java Object Header Elimination for Reduced Memory Consumption in 64-bit Virtual Machines. *ACM Trans. Archit. Code Optim.* 4, 3, Article 17 (Sept. 2007). https://doi.org/10.1145/1275937.1275941

[17] Christian Wimmer. 2004. Linear scan register allocation for the Java HotSpotTM client compiler. *Master's thesis, Johannes Kepler University Linz* (2004).

[18] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[19] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2008. Visualization of Program Dependence Graphs. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 193–196. http://dl.acm.org/citation.cfm?id=1788374.1788391

[20] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013)*. ACM, 187–204.