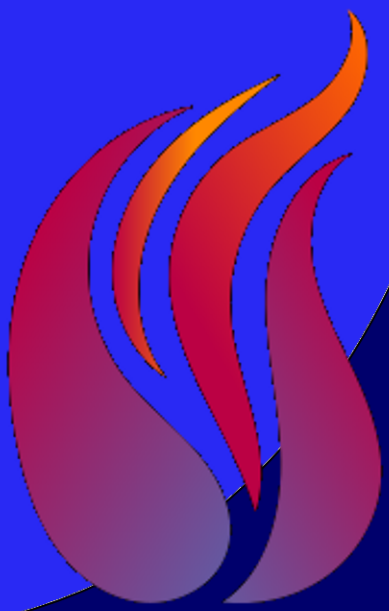


JAA

JAA: JVM Arm Accelerator Technical Report

Seyed Iman Hosseini Zavaraki

Contents

1 INTRODUCTION

JAA: JVM ARM Accelerator is a project with the aim of translating JVM bytecode to ARM instructions for the purpose of accelerating execution. JVM runs on many different platforms, including ARM-based systems and such translation to native ARM code can make code execution faster, and JAA aims to do this via Verilog HDL; thus it is possible to synthesize it as a co-processor for ARM processors.

There has been past efforts to do such low level code translation in the form of Jazelle extension by ARM. [1] There is also related work in literature and projects involving tweaks to the Just-in-time compiler to enhance performance of JVM [2][3] and a patent for a coprocessor for software execution acceleration. [4]

We begun our work by conducting a review of prior work, and researching ARM and JVM manuals. We realized early on that in order to make a real advance, we need a solid infrastructure and we need to design it ourselves. With the aim of scalability and extensibility in mind, we designed JAA's infrastructure to be a platform for future work and to be easy to extended by the open source community. Straight off, we wanted a vertical slice of what is achievable because it does not make sense to have a code which generates something we cannot actually execute. The details of design and how we achieved these goals are discussed in the following sections of this report.

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0035b/BHBEIBIF.html

[2] http://www.openjit.org/

[3] Matt Welsh and David E. Culler. "Jaguar: enabling efficient communication and I/O in Java". In: *Concurrency - Practice and Experience* 12 (2000), pp. 519–538.

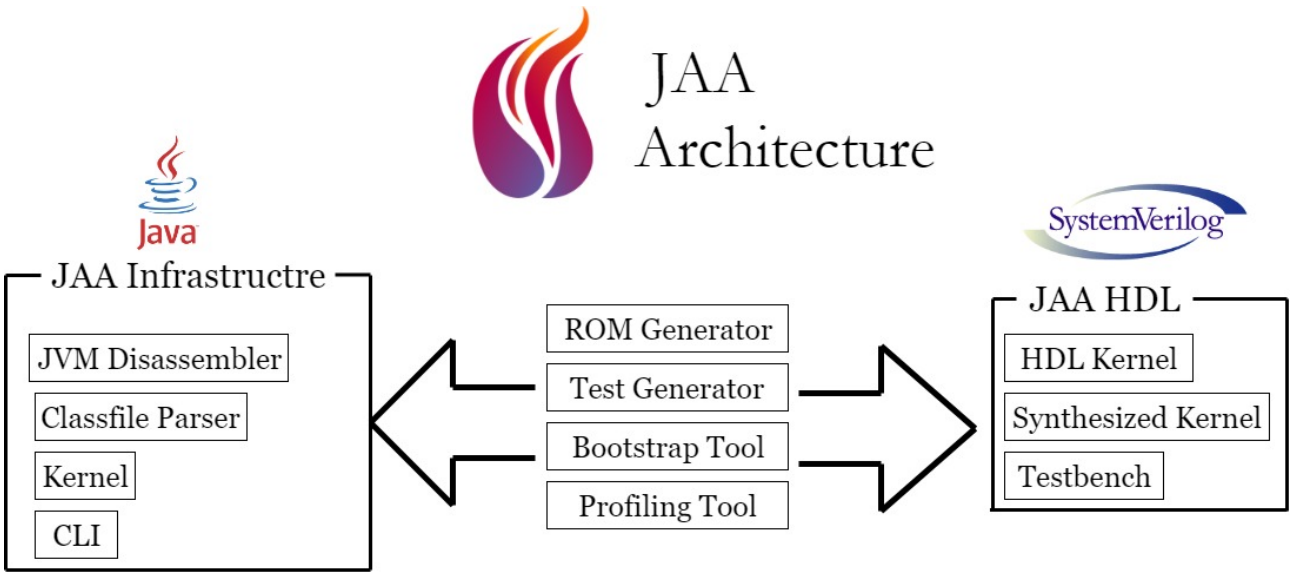
[4] https://patents.google.com/patent/US20160077852A1/en

2 JAA ARCHITECTURE

2.1 TOPLEVEL VIEW

JAA was created with the aim of serving both as a working example, and also as infrastructure for future work. We wanted JAA to be readily updatable with latest changes to ARM and JVM specifications in future versions. Also, as we could not find a similar infrastructure for our work, we wanted to make our tools accessible for similar bodies of work, as an example one can easily extend our ”Instruction” class to add support for other machines (e.g. x86).

JAA consists of three main modules, infrastructure, HDL and the interfaces between them. The infrastructure is based on Java, the HDL is the Verilog implementation and the result of synthesis, and the interface consists of Java tools which expose the HDL part directly. Each of these parts, are made up off smaller modules.



JAA ARCHITECTURE

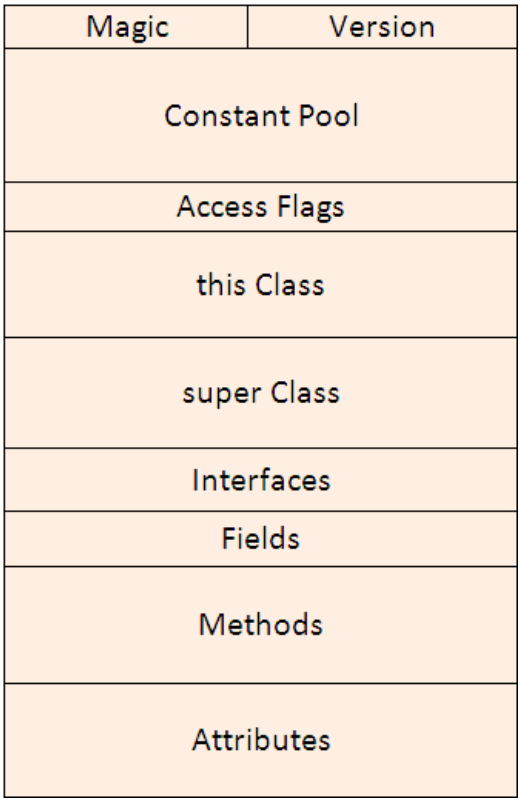
One of the design decisions was implementing different functionalities which were already implemented in other software packages from scratch, or integrate third-party code to our codebase. An example is Apache BCEL (Byte Code Engineering Library) [a] which included a tool which parsed Classfiles that we needed, but we decided against it and implemented our own parser from scratch for two reasons: 1) It was a small part of the whole BCEL project but using it would bring other dependencies to our project. 2) The parser we needed was simpler, and thus we were able to implement an optimized version with less overhead that the BCEL solution.

[a] https://commons.apache.org/proper/commons-bcel/

2.2 .JB FORMAT

As indicated by prior work, and our own research the point of such acceleration is not to cross-compile the whole set of instructions, but only the arithmetic. Evidently, the whole translation of all JVM would be infeasible and also circumvent JVM completely.

So if we wanted to test our generated native code, we needed a setup to provide us with the runtime, and also a way to only tackle the arithmetic part. We developed a file format, JAA Binary (.jb) which is a simplified classfile. The format of classfile, is complicated and includes many sections not needed by our kernel.



CLASS FILE FORMAT

When a java program is developed it is composed of .java files including the source files, which are then compiled using javac, into .class files. These files are in a way, similar to .o object files of C, and include the bytecode and metadata which then runs on the JVM. Other languages using JVM, like Kotlin, also have a similar procedure and their source compiles to .class files.

What we need from a classfile, is the code of main method, and the fields in Constant Pool which include Double, Long, Integer and Float constants. Also we need to maintain a byte off-set table, as .class files are parsed sequentially and include data structures with different sizes for reference of our kernel (which does the main cross compilation).

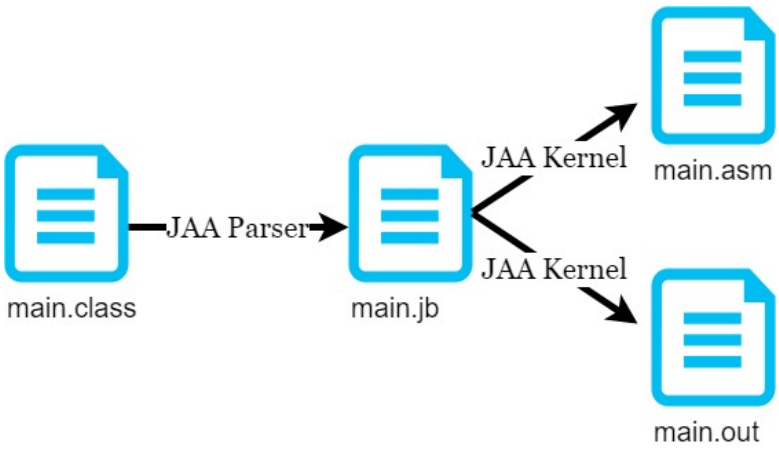
.jb file format

4 bytes	0xCAFEFAAA	Magic Number
2 bytes	Version	short specifying version number
2 bytes	cp_data offset	First byte of cp_data
2 bytes	code offset	First byte of code
X bytes	cp_table	Table containing constant pool table
X bytes	cp_data	Constant pool data
X bytes	Code	JVM bytecode of main method

JB FILE FORMAT

Our .jb format does just that, and we have a module in JAA Infrastructure for that: The Classfile Parser. This module transforms a .class file into our .jb file, which is a stripped down version of the initial classfile, and keeps the bytecode intact. Then, our kernel (which can be the Java version or the HDL version: Verilog simulated, or the synthesised version) would process that .jb file and output in two format, native ARM binary and ARM assembly format (which can be assembled into native binary via ARM assembler).

Making a binary without the use of ARM assembler, proved to be a big challenge, as we had to somehow do the job of an assembler which is a contrived task on its own. Also, an assembly files would suffice for our testing purposes, as we could then assemble it and compare its execution with the original java program to validate and profile our algorithm. But in the real scenario we would want to incorporate a coprocessor to an ARM processor, and thus we wanted to see how the binary-to-binary actually functioned, particularly interesting would be the timing analysis of that.



DATAFLOW

3 IMPLEMENTATION

3.1 JVM INSTRUCTIONS

Based on Oracle's bytecode listing, we have already implemented 19 instructions (each seperate entry in the bytecode listing is counted as one instructions, e.g. iload has 5 instructions, iload, iload_0,.. ,iload_3) the bytecode classes are iload, istore, iconst, iadd, return. Work is underway for other instructions as well, including instrutions for subtraction, multiplication and division. And also to support floating point versions of these bytecodes. Also there are conversion instructions like d2f (converts double to float) which are being worked on. The total number of instructions surveyed is more than 40 instruction, which we hope to implement at least 24 of them in this portion of project.

3.2 JAA INFRASTRUCTURE AND HDL INTERFACE TOOLS

The infrastructure is implemented in (you guessed it) Java. The JAA tools is developed in a manner respecting the general OOD guidelines to be scalable and extendable. It also includes an executable jar, which is a CLI interface to the core functionalities, for those who do not wish to get their hands dirty with the source code.

3.2.1 KERNEL & UTILS

3.2.2 TEST GENERATOR

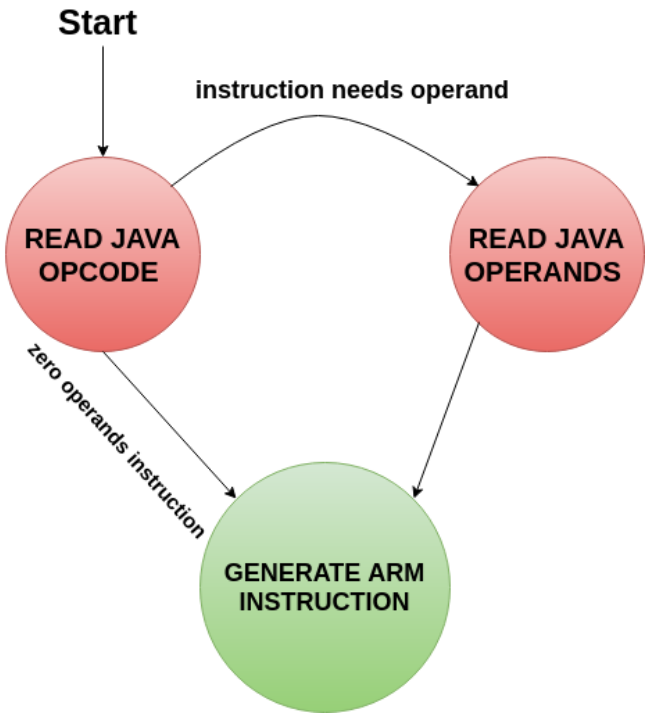
3.3 JAA HDL

The HDL module, implemented in Verilog, is fairly simple. It is a glorified decoder (a big switch case). As can be seen from the state machine, the module reads the .jb program from a ROM and translates the instructions like an interpreter would.

4 PERFORMANCE ANALYSIS

4.1 METHODOLOGY & RESULTS

To evaluate the performance of JAA, we conducted tests on a Raspberry Pi 3. To do a comparison between what JVM does, and what JAA does, we needed a robust method. We devised a simple class of model programs with a parameter to control the size of the program, and then made it into a .class file and ran it using java command. And for running it via JAA, we produced native code using JAA and then ran the resulting native executable. Notice that the following scheme also works as a way to verify the results, by echoing the return code (this would be the



STATE MACHINE

r0 register, so we can measure it against the final result of a computation in the java code).

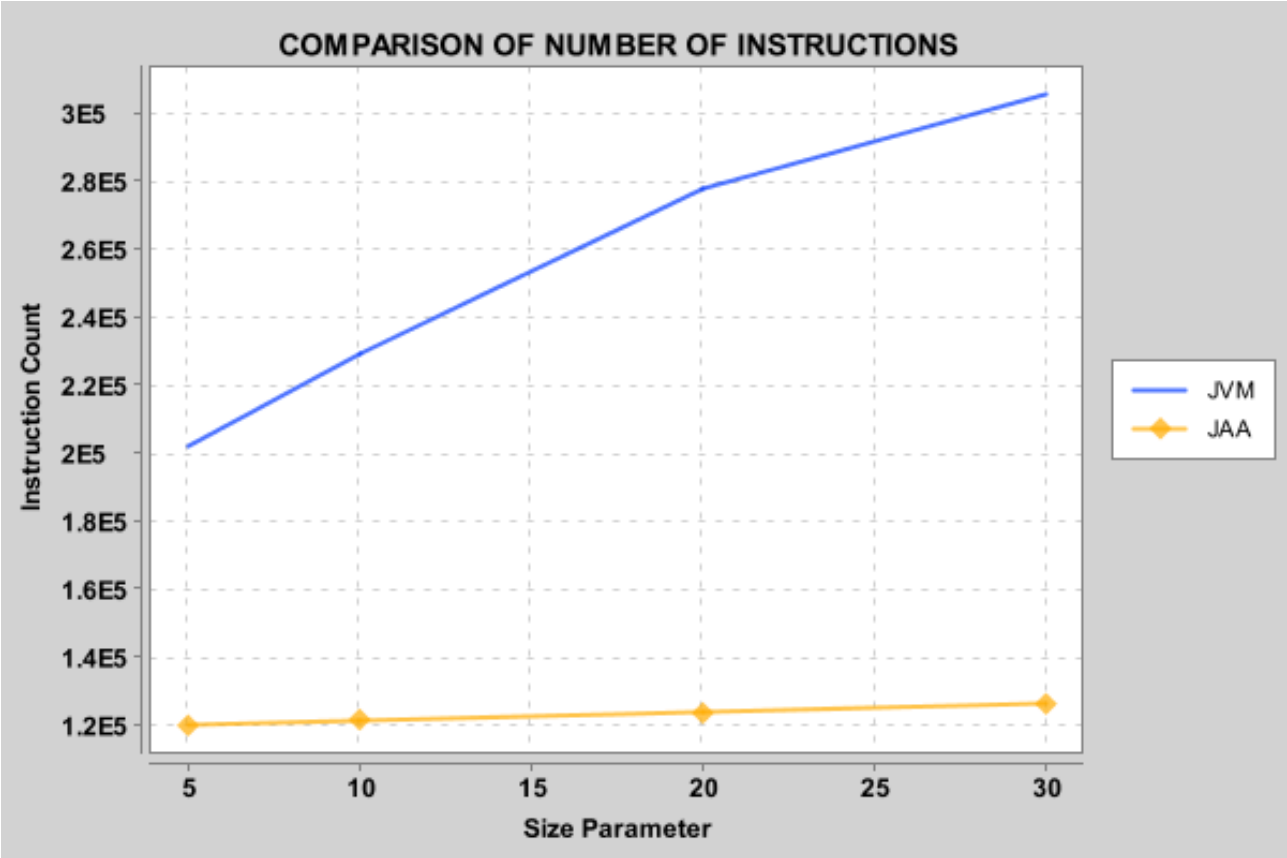
There was a missing link though: analyzing the time spent on each task, is not robust. It is prone to random fluctuations and very much depends on the configuration of the system at the time. To counter this, we used Valgrind tool[5] . Valgrind is a framework for debugging and profiling Linux programs. It has been used for research purposes by people at many universities among them: Cambridge, MIT, UC Berkeley. Among the it's tools, there is a tool called 'callgrind' which we have used. This tool, reports the number of instructions executed by the program under test, thus eliminating the non-deterministic nature of time measurement. For this reason, this is the standard method of comparison in research.

Our model programs are very simple, we call it the N-P model. An N-P program has N variables which are sequentially updated with random arithmetic operations in P consecutive passes. It is a trivial fact that the size of the program with respect to N, P is linear.

The initial results though were still an unfair comparison, because the total number of instructions returned by running java, included the instructions which managed setting up the JVM. So we calculated the number of instructions for the case of a program with only a return statement, and subtracted that from our results.

As evidenced by the evaluation results, JAA indeed surpasses JVM. There is a speedup fac-

[5] https://valgrind.org



JAA VS JVM

tor between 2x, 3x which proves the viability of our approach. As outlined in the introduction, without testing the output of JAA on a real Arm based system, testing and evaluation would not be possible and any 'synthesized' test would have been pointless, and we put great effort to have an end result that actually runs on a Raspberry Pi.

4.2 LIMITATIONS

The main limitation as of now, is that we haven't yet coevered all of JVM instructions.