Progress Report

Java Virtual Machines for Heterogeneous Multicore Architectures

Performing an Implementation Research of Maxine VM on the ARMv8-A 64-bit Architecture

MSc Student: Jiaqi Liu
Supervisor: Mikel Lujan

May 2015
# Table of Contents

Abstract....................................................................................................................... 3

1 Introduction ........................................................................................................... 3
   1.1 Project Context................................................................................................. 3
   1.2 Aim and Objectives......................................................................................... 4
   1.3 Deliverables........................................................................................................ 5
   1.4 Report Structure.............................................................................................. 5

2 Background ............................................................................................................ 5
   2.1 Java Virtual Machines and the Maxine VM..................................................... 5
   2.2 Interpreters and Compilers............................................................................. 7
   2.3 T1X, C1X and Graal ....................................................................................... 7
   2.4 ARMv8 Architecture....................................................................................... 10
      2.4.1 Registers.................................................................................................. 10
      2.4.2 Stack Frame............................................................................................. 12
   2.5 Related work .................................................................................................... 13

3 Research Methods .................................................................................................. 13
   3.1 Methodology – An Incremental Approach...................................................... 14
   3.2 Development Environment Setup ................................................................. 15
   3.3 Project Evaluation ......................................................................................... 16
   3.4 Maxine Modules.............................................................................................. 16

4 Progress ................................................................................................................... 17
   4.1 ARMv8 Platform Abstraction ....................................................................... 17
   4.2 ARMv8 Representation .................................................................................. 18
   4.3 ARMv8 Representation Evaluation ................................................................. 20
   4.4 Next Step......................................................................................................... 20

5 Plan and Coming Tasks .......................................................................................... 21

6 References............................................................................................................... 22
List of Figures

Figure 1-1 JVMs on different platforms ................................................................. 3
Figure 2-1 Structure of the Maxine VM. This project involves the optimising compiler Graal, the T1X compiler, the assembler and the substrate VM boot loader. ........................................... 6
Figure 2-2 Templates used by T1X compiler .................................................................. 8
Figure 2-3 Graal reads bytecode, optimises the code in different levels and finally generates target machine code........................................................................................................... 9
Figure 2-4 Optimising and De-optimising between T1X and C1X/Graal.......................... 9
Figure 2-5 Registers in x86_64 and ARMv8.................................................................. 11
Figure 2-6 Stack frame of ARMv8. Frame pointer and stack pointer point to different locations on the call stack........................................................................................................ 12
Figure 3-1 Overall workflow .......................................................................................... 14
Figure 3-2 Project Evaluation .......................................................................................... 16
Figure 4-1 UML about ARMv8 representation................................................................. 17
Figure 4-2 Register definition class ................................................................................ 18
Figure 4-3 ARMv8 register groups ................................................................................. 19
Figure 4-4 Creating VM ................................................................................................. 20
Figure 4-5 Stubs and Trampoline in Maxine ................................................................... 21
Figure 5-1 Project plan ................................................................................................. 22
Abstract

Java virtual machines (JVMs) enable programming languages, not only Java, to be executed on various platforms without modification. Among many prominent JVMs, Maxine provides an excellent and extensible research platform for high level managed languages and their compilation, due to the proactive use of software abstraction and a highly modular structure.

This project is to implement Maxine on the state-of-the-art ARMv8 64-bit processor, which is particularly attractive and can be of great help for future research. This implementation involves three major modules: a unit test framework for compilers, the T1X baseline compiler and the Graal JIT compiler.

This report gives a detailed progressive description of the research on implementing the Maxine VM on ARMv8, including its research context, design and evaluation. In addition, a plan of the whole project lifecycle and tasks for the next stage are also discussed.

1 Introduction

1.1 Project Context

Java, referred to as both a programming language and a software platform, is one of the most popular and successful software systems [1] [2]. While big data processing is gaining its importance nowadays, Java also plays an important role in this trend. Distributed data processing software frameworks written in Java, such as Apache Hadoop, have proven their power in high-performance computing over a large scale of data [3].

![JVMs on different platforms](image)

The popularity of Java mostly owes to its feature of being platform independent and robust. The runtime environment of Java has been implemented on various CPU architectures and operating systems, which gives Java programs great portability. As the core part of the Java runtime, Java virtual machines (JVMs) execute Java bytecode, usually compiled from Java source files, on different platforms and guarantee the
correctness of the execution. Figure 1-1 shows that different languages can be compiled to Java bytecode and then be executed by JVMs on different platforms.

In the last two decades, many JVM implementations have been implemented [4]. Among them, the Maxine virtual machine is mainly targeted at the research for Java language and compilers. Maxine is entirely written in Java (a metacircular JVM) with very high level of abstraction [5]. It is possible to use whatever optimising compilers as long as they have implemented the compiler-runtime-interface (CRI, a compiler interface that defines how a compiler should behave in order to be used in different JVMs). Thus, Maxine provides an ideal environment for high level language and compilers research.

The ARM architecture is a reduced-instruction-set-computing (RISC) architecture. In terms of the number of annually shipped processors, it accounts for the largest microprocessor market share in recent years [6]. Most handheld devices, including smartphones and tablet PCs, are equipped with ARM processors. Meanwhile, ARM processors provide relatively better energy efficiency compared with complex-instruction-set-computing (CISC) ones. Recently, 64-bit ARM architecture (ARMv8 or AArch64) has been introduced [7]. Together with the ARM big.LITTLE heterogeneous architecture [8], which was announced in 2011, ARMv8 provides a more efficient way of energy saving computing.

However, a problem appears if researchers are looking for a JVM for Java language research on the ARMv8 architecture. The Maxine VM has no official support for ARMv8 (only support for the x86_64 architecture is officially provided).

To gain knowledge of the performance of just-in-time (JIT) compilers and managed runtime environments, it is important to investigate them on state-of-the-art processor platforms. Therefore, the necessity for porting Maxine to ARMv8 is clear. Once succeeding in the port, developers will be offered an excellent opportunity for the research of compilers and managed runtime environment on ARM's 64-bit architecture.

1.2 Aim and Objectives

Simply stated, the aim of this project is to implement the Maxine virtual machine on the ARMv8 architecture. As will be discussed in detail in section 2, Maxine consists of well-defined modules that make it possible to be ported to other architectures other than x86_64.

To be more detailed, the objectives of this project are listed as below:

- Setting up an environment for the Maxine VM and the Graal compiler that is needed as the optimising JIT compiler in Maxine. (The combination of Maxine and JIT compilers will be introduced in later sections.)
- Implementing ARMv8 architecture representation in Maxine and Graal source code. The representation consists of register organisation and architecture features representations, such as endianness and memory model.
• Porting unit test framework which is implemented in a well-advanced ARM 32-bit Maxine port. The unit test framework activates an offline compiler to compile code fragment and then verifies the generated target code.

• Porting the Graal optimising compiler and T1X baseline compiler. This involves implementing the ARMv8 assembler for Graal and T1X, which is used to convert Java bytecode instructions to target ARMv8 machine code.

1.3 Deliverables

The deliverables come in the following forms.

• Ported Maxine and Graal
  The ported modules of Maxine and Graal will be delivered in forms of source code along with their Mercurial (a version control tool) commit history. A runnable Maxine VM image will also be delivered.

• A document on porting details
  A detailed implementation and porting description are created as the project is being carried out and will be delivered to help future study.

• A document on environment setup and Maxine usage
  This document describes how to setup the runtime environment and the development environment, also in order to help future investigations.

1.4 Report Structure

Section 2 further discusses a wider context of generic JVMs, Maxine VM. Introduction to ARMv8 64-bit architecture and comparison of ARMv8 and x86_64 are also provided. As part of the technical context, related JVM porting practice and previous work are reviewed in this section to give a clear guideline on how to implement Maxine on different architectures.

Section 3 provides a gradual methodology of porting Maxine and the setup of the development environment. A modular analysis of Maxine’s structure is then discussed followed by the evaluation method of the project.

Section 4 describes the current progress of the port and its evaluation.

Section 5 gives a plan to guide the project’s execution.

2 Background

2.1 Java Virtual Machines and the Maxine VM

Ranging from handheld devices to large scale data processing servers, Java’s application in different areas has significantly promoted modern software world. JVMs have contributed to this success by providing Java the cross-platform feature. As is shown in Figure 1-1, JVMs have been implemented on various architectures. They take Java bytecode as input and guarantee the correctness of the execution on different platforms.
A running JVM is actually a process which reads the bytecode instructions and interpret or compile them into target machine code. More details about JVM interpretation and compilation will be discussed in section 2.2.

The behaviours of JVMs are defined by the Java virtual machine specification [9]. Not caring about how to implement a JVM, the specification only cares about what to implement in order to be qualified as a JVM. As a result, numerous JVMs have taken place on various platforms, Oracle Hotspot VM, Maxine VM and Apache Harmony VM to name but a few.

The aim of most JVMs is to obtain maximal execution performance. Taking Hotspot as an example, to achieve this goal, the source code of Hotspot is relatively complex with many optimisation tricks that often confuse high level language researchers [10]. Adding an extensional feature or function in such a complex software system is often time-demanding and error-prone.

![Diagram of Maxine VM structure](image)

**Figure 2-1 Structure of the Maxine VM. This project involves the optimising compiler Graal, the T1X compiler, the assembler and the substrate VM boot loader.**

The Maxine VM, on the contrary, is mainly aimed at language and compiler research. Maxine is entirely written in Java, which can be called as a meta-circular JVM (Java programs executing on a JVM written in Java). It has a very high level modular design with a clear logic of how different modules of the VM work together. Figure 2-1 shows an overall structure of Maxine. The top part of the figure consists of the Java bytecode processing modules. Java class files (bytecode) are firstly loaded and verified. Then they
are compiled by the baseline compiler (T1X) or by the optimising JIT compiler (Graal or C1X depending on given parameters). These compilers will be further discussed in the next section. The lower part of the figure consists of modules that manage the execution of compiled code. Memory management (with garbage collection) and thread scheduling are two major modules. Written in Java, Maxine is equipped with the ability of utilising very high level abstraction and aggressively decoupled design [5]. Its coverage of the JVM specification is nearly complete which makes it compatible with Java bytecode compiled from standard Java compilers.

With these features, Maxine is an ideal platform on which developers can carry out managed language and compiler research since it is feasible to write extensions and extra add-on modules for such a highly modular virtual machine.

2.2 Interpreters and Compilers

When the first version of Java Development Kit (JDK 1.0) was released in 1996, it only provided a pure interpretive execution virtual machine (Sun classic VM). In other words, at that time Java bytecode could only be interpreted, not compiled [11]. As a result, Java suffered from a performance penalty, often much slower than native languages such as C or Pascal [12].

To improve Java's performance, in 1998 a virtual machine named Hotspot was released along with JDK 1.2 and then later became the default VM for JDK and JRE (Java runtime environment). The Hotspot VM comes with a just-in-time (JIT) compiler named C1, which can compile frequently executed code (hotspot code) into optimised machine code in order to avoid repeated interpretation.

However, interpreters are not abandoned. One important reason is that converting Java bytecode to machine code via interpretation is faster than via compilation (although the interpreted code is often slower because of lack of optimisation). Thus, to minimise the loading time of a piece of a program, an interpreter is used to convert the bytecode instructions into machine code without having to wait for the optimisation that a JIT compiler always performs.

After a piece of Java bytecode is loaded by the JVM, it is firstly interpreted and then executed. If later this code is detected as hotspot code (repeatedly executed), the JIT optimising compiler will compile it into optimised machine code. According to [13], hotspot code accounts for 80% - 90% of a program's execution time while for only around 20% of the code size. So it is effective to only compile hotspot code because it gives a good balance of loading time and executing time.

2.3 T1X, C1X and Graal

The Maxine VM does not use an interpreter. However, a template-based baseline compiler called T1X resembles the role of an interpreter in Maxine. Although acting as a compiler, T1X seeks a maximal speed of machine code generation [5] [14]. Templates for bytecode instructions are used to maximise the efficiency of this process. Templates
consist of predefined machine code for each bytecode instruction (see Figure 2-2). Compiling bytecode via T1X simply means copying these prebuilt machine code instructions to the code buffer. Then, the machine code in the code buffer will be combined with other machine code that may be compiled by T1X or other compilers.

Due to the simplicity of T1X, it has the ability of being easily extended. For example, if researchers want to test a novel translation of a bytecode instruction, they no longer bother to orchestrate a series of complicated sequences in the optimising JIT compiler. All they need to do is to write a template, or even to replace current corresponding template, in T1X and then test the compilation result. As illustrated in Figure 2-2 creating or replacing entries in T1X is very straightforward because these templates are highly independent (decoupled).

```java
@T1X_TEMPLATE(IUSHR)
public static int iushr(@Slot(1) int value1, @Slot(0) int value2) {
    return value1 >>> value2;
}

@T1X_TEMPLATE(IRETURN)
@Slot(-1)
public static int ireturn(@Slot(0) int value) {
    return value;
}

@T1X_TEMPLATE(IRETURNUnlock)
@Slot(-1)
public static int ireturnUnlock(Reference object, @Slot(0) int value) {
    Monitor.noninlineExit(object);
    return value;
}

@T1X_TEMPLATE(ILOAD)
public static int iload(@Slot(1) Object array, @Slot(0) int index) {
    ArrayAccess.checkIndex(array, index);
    int result = ArrayAccess.getInt(array, index);
    return result;
}
```

**Figure 2-2 Templates used by T1X compiler**

However, if only relying on T1X, Maxine will not be able to obtain satisfactory performance. As a result, optimising JIT compilers are used to generate high-quality machine code. Currently, Maxine can utilise C1X and/or Graal as its optimising compiler.

The C1X compiler is a variation of the optimising client compiler (C1) built in Hotspot VM, which is introduced in section 2.1. Mainly focused the optimising target code, C1X is designed entirely different from T1X and is more complicated. A lot of optimisation strategies are exploited, such as dead code elimination, global value numbering and block merging [5] [15].

The Graal compiler was firstly created as a “Java port” of C1 compiler (C1 is mostly written in C++). Using Java, Graal is able to aggressively use high-level abstractions and
highly modular structures (a similar situation happens to Maxine as talked before). This makes Graal easier to adapt new compilation improvements.

Figure 2-3 Graal reads bytecode, optimises the code in different levels and finally generates target machine code.

Graal can be used by Maxine in a standalone mode or in a mixed mode with C1X as the failsafe compiler. The combination of Graal and Maxine is shown in Figure 2-3. Notice that Graal uses different levels of intermediate representation (IR) in different compilation stages. In this project, we only focus on the last two stages (machine code generation and machine code deployment) in the figure.

Due to Graal’s extensible feature, we select Graal as the optimising JIT compiler in our project. So this project involves porting work for both T1X and Graal.

Figure 2-4 Optimising and De-optimising between T1X and C1X/Graal
Most modern JVMs use a combination of an interpreter/baseline compiler and an optimising JIT compiler. The reason is introduced in section 2.2. The interaction between interpreters/baseline compilers and optimising JIT compilers are illustrated in Figure 2-4. When a piece of bytecode is determined as hotspot code, it will be compiled by the optimising JIT compilers. JIT compilers perform aggressive optimisations to the code. However, applying these speculative optimisations requires some common hypothesis about the runtime environment and objects. Sometimes, the compiler may find the hypothesis cannot be satisfied, for example, new classes are loaded or the class hierarchy structure is changed. That is called the “uncommon trap” [16]. In this case, the compiler will perform de-optimising operations within which the bytecode is passed back to the interpreter/baseline compiler. Since interpreters/baseline compilers do not perform speculative optimisations, they can produce correct but usually slower machine code, acting like a failsafe procedure.

To summarise, the clearness work flow of T1X and the modular structure of Graal inspire us to implement our project with an incremental approach in which each module is implemented independently in a well organised order. This approach is detailedly discussed in section 3.

2.4 ARMv8 Architecture

As introduced in section 1.1, the ARMv8 architecture has the support for 64-bit operation and addressing. Compared to previous ARM 32-bit architecture, it also provides more and larger (64-bit) general purpose registers (r0 – r30) and floating point/SIMD registers (v0 – v31). SIMD (single instruction, multiple data) registers can be access as vector registers whose contents are operated in a parallel way to boost performance (details can be referred to in [17] and [18]). All these features give ARMv8 the ability of managing more memory and accomplishing high-performance operations.

To perform research of Maxine VM and Graal compiler on ARMv8, the basic structure and concepts of ARMv8 should be studied first. In the rest of this section, the registers and the stack frame will be discussed.

2.4.1 Registers

Register allocation is a critical aspect when a compiler generates machine code. As register are much faster than memory, efficient use of registers can significantly boost the performance of programs. Therefore, in this section the features of ARMv8 registers are studied in order to consolidate the VM/compiler level implementation of ARMv8 architecture. As will be seen in section 4, the

As shown in Figure 2-5, compared to x86_64 architecture, ARMv8 provides more registers that are available to developers. The usage of ARMv8 general purpose registers is listed in Table 2-1. These general purpose registers are served as integral registers dedicated to scalar integer computation and memory addressing. In integer computation mode, these registers can be used in 32-bit and 64-bit while in addressing mode they can only be used in 64-bit mode.
One noticeable thing is that the stack pointer (SP) can only be accessed in limited situations:
• Store and load instructions
  The stack pointer has to be quad-word aligned (aligned to 16 bytes). If not, a "stack alignment fault" will be issued.
• Add and subtract instructions
  In this case, the stack pointer can be used as the pointer to the location of data.
• Logical data processing instructions
  Same as item 2, stack pointer is used to indicate the location of source or destination addresses.

In addition to the general purpose registers mentioned above, ARMv8 provides 32 128-bit floating point/SIMD registers (v0 – v31). For single and double precision floating computation, they can be accessed in 32-bit or 64-bit (similar to general purpose registers). For "single instruction, multiple data" instructions, they can be used in 64-bit or 128-bit mode.

The implementation for register representation can be referred to in section 4.2.

2.4.2 Stack Frame

ARMv8’s stack implementation is full-descending. When a “push” operation happens, the stack pointer decreases, which means the stack goes to the lower address of memory when its size increases. Within the stack, each element must be quad-word aligned (64-bit).

![Stack Frame Diagram](image)

**Figure 2-6 Stack frame of ARMv8. Frame pointer and stack pointer point to different locations on the call stack.**

An important concept in Java method calling is the call stack frame which is heavily used in Maxine/Graal compiler [19]. Figure 2-6 shows the basic structure of stack frames in a call stack.
One of the most common operations in the programming world is calling another method or function. To preserve the context of the caller and callee, a call stack is created to save runtime information. It is usually divided into different but continuous blocks which are named as stack frames. As shown in Figure 2-6, when a caller calls another method (function), a new stack frame is created. Now the stack pointer points to the next free space on the stack top. But a frame pointer (FP) is used to point to the beginning of the current stack frame. The current stack frame contains runtime information of current active method/function. Thus, using the frame pointer the system can acquire the current running state which is also useful for debugging and compilation optimisation.

2.5 Related work

Maxine’s implementation on 32-bit ARM architecture

Researchers from the APT group (the Advanced Processors Technologies Research Group) [20] of the University of Manchester have been working an implementation of Maxine on the ARMv7 32-bit architecture [21]. This port is well advanced. The substrate, the assembler, T1X, C1X and the substrate have been ported, and the bootstrap is currently being debugged. This port has developed a unit test framework for testing the optimising compiler, the template compiler T1X and the assembler without having to boot the whole JVM.

Our ARMv8 Maxine project considerably benefits from this ARMv7 Maxine implementation. For example, as will be described in section 3.1, the unit test framework developed in ARMv7 Maxine can be ported to fit in our project to perform efficient evaluation.

Jikes RVM’s implementation on 32-bit ARM architecture

Jikes RVM is another JVM developed in Java itself [22]. Similar to Maxine, Jikes RVM is also targeted at providing a high level language research platform.

In [23], an ARMv7 Jikes RMV porting activity is described. Although not able to execute a complete Java program, this port has implemented the compilation of most Java bytecode instructions. In addition, it offers a good methodology in which minimal changes are made to implement each module.

3 Research Methods

As described in section 1.2, the final aim of this project is implementing the Maxine VM on ARMv8. In this section, the methodology and steps of achieving this ultimate aim will be discussed in detail. The overall model of how this research is scheduled is illustrated in Figure 3-1.
3.1 Methodology – An Incremental Approach

As Maxine and Graal are well defined in a modular scheme, an incremental approach can be used to perform Maxine’s implementation on ARMv8. In this approach, we are enabled to focus on one single topic at each stage with fewest modules involved, which can help to make the implementation clearer and less error-prone.

This incremental approach is described as follows:

- Analyse and distil the x86_64 representation in Maxine and Graal
- Track the initialisation stage in Maxine to get proper configuration of x86_64 mode compilers
- Implement ARMv8 representation
- Reconfigure settings for x86_64 mode compilers to activate offline Graal in x86_64 mode
- Track the x86_64 offline Graal compilation stages
- Implement ARMv8 offline Graal compilation stages
- Porting unit test framework from the 32-bit ARM port of Maxine
- Porting Graal and T1X online compiler
- Implementing JVM boot loader on ARMv8/Linux platform

As can be seen from the above items, the existing x86_64 architecture support is mostly used as the reference to the ARMv8 implementation. By tracking and debugging the x86_64 implementation, it is easier to perform research and design on ARMv8.

Each of the above items is independent to the latter items and involves fewest possible modules. Moreover, the above items can be used as a detailed plan for the project since each of them can be achieved and evaluated with clear schedules. In other words, each item is regarded as a subgoal which can be fitted into the overall work model shown in Figure 3-1.

What is more, with each ongoing subgoal, an accompanying document will be created to record 1) the activities that have been done and 2) easy-to-understand description of programming outcomes in the ongoing subgoal. The purpose of keeping trackable documents is to help debug and revise the project as well as to make possible future research easier to follow the current work.

### 3.2 Development Environment Setup

This section describes the setup of the environment needed by the research of implementing Maxine on ARMv8.

To carry out this project, the host platform and toolchain are shown as followings:

- **x86_64 GNU/Linux (Ubuntu 14.04 LTS)**
  64-bit Linux is needed to build Maxine. Long-term support (LTS) version of Ubuntu is selected to ensure the stability of the host platform.
- **Oracle JDK 7 update 25**
  Maxine is built with Oracle’s official JDK version 7. Due to some minor but critical differences between Open JDK and Oracle JDK, Maxine is not fully compatible with Open JDK.
- **GNU Compiler Collection 4.8.2 (gcc/g++)**
  Building the substrate (JVM boot loader) needs to use GCC. This version is the default one built in Ubuntu 14.04 LTS.
- **Eclipse**
  Eclipse is used to write most of the Java code. An integrated development environment (IDE) like Eclipse is useful especially when handling a complicated software system.
- **ARMv8 Emulator (based on QEMU)**
QEMU [24] is a machine emulator with which ARMv8 can be emulated. The generated machine code is tested with ARMv8 QEMU to verify the code’s correctness.

3.3 Project Evaluation

The evaluation of this project comes in different ways (see Figure 3-2).

<table>
<thead>
<tr>
<th>Category</th>
<th>Subgoals</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv8 Representation</td>
<td>Register Representation</td>
<td>VM boot-up initialisation</td>
</tr>
<tr>
<td></td>
<td>VM Configuration</td>
<td>VM boot-up initialisation</td>
</tr>
<tr>
<td>Offline Graal</td>
<td>Stubs/Trampolines</td>
<td>Java prototype verification</td>
</tr>
<tr>
<td></td>
<td>Basic Assemblers</td>
<td>QEMU Emulator Execution</td>
</tr>
<tr>
<td>Graal</td>
<td>Stack Frame</td>
<td>Unit test/QEMU</td>
</tr>
<tr>
<td></td>
<td>Register Allocation</td>
<td>Unit test/QEMU</td>
</tr>
<tr>
<td>T1X</td>
<td>Templates</td>
<td>Unit test/QEMU</td>
</tr>
</tbody>
</table>

Figure 3-2 Project Evaluation

Register representation, VM configuration can be evaluated by booting Maxine and verify the debug information to see if the VM can be successfully booted. The outcome of activating offline Graal is evaluated by the Java prototype initialisation/verification and testing the generated machine code on QEMU ARMv8 emulator. Similar machine code evaluation on QEMU is used for (online) Graal and T1X following unit tests for each staged process.

3.4 Maxine Modules

Shown in Figure 2-1, the highlighted parts are involved in this project.

Maxine is especially configurable. Maxine uses schemes to define possible behaviours of how one module should communicate with other modules within the VM. Schemes are actually Java interfaces and abstract classes that can be implemented to achieve certain goals such as memory management, thread scheduling and machine code generation. This structure gives an exceptional capability to be extended with other features, for example to obtain different optimising levels in different debugging environments.

With this advantage, we can create compilation modules based on these schemes by implementing interfaces or extending abstract classes. The following section discusses how we abstract and implement the ARMv8 architecture.
4 Progress

4.1 ARMv8 Platform Abstraction

Simply stated, we have implemented the ARMv8 representation and activated ADM64 offline compiler. In Maxine and Graal, the term "ARMv8" stands for "x86_64 architecture". In accordance with that, we use "AARCH64" (ARM Architecture 64-bit) to represent "ARMv8".

![UML diagram about ARMv8 representation](image)

To illustrate the current progress, a simplified UML chart is shown in Figure 4-1. In order to focus on the basic structure, the redundant class fields and methods are removed from the chart.

Class *MaxineVM* is the entry class of the virtual machine. As can be seen from the dashed arrow lines (which means dependency in UML) in Figure 4-1, MaxineVM requires platform definitions such as *AMD64* (x86_64) and *AARCH64* (ARMv8). The AARCH64 class, which will be discussed further in the next section, is used to define the register structure and platform features of ARMv8, such as memory endianness (byte order) and memory barriers.

Class *RegisterConfigs* holds the configuration for each platform and is needed to define register allocation rules, such as caller/callee register layout and parameter passing layout. It reads platform specific register behaviour and layout requirement to generate
correct configurations. In the bottom of Figure 4-1, AARCH64SafePointPoll reads AARCH64 to get the register layout and then determine the safe points (in safe points the VM can trigger actions like garbage collection without hazard.)

4.2 ARMv8 Representation

Register representation, written in class AARCH64, is the central part of ARMv8 implementation. All later steps are based on how registers are represented in Maxine and Graal. Registers are grouped into three categories because in different contexts they are used for specific purposes. For example, the integral registers are mainly used to pass parameters, convert addresses and perform logic calculations while floating point/SIMD registers are targeted at floating point calculation and even cryptography computations. The brief register class constructor definition is shown in Figure 4-2 and the grouping code is shown in Figure 4-3.

In Figure 4-2, the properties of a register class can be easily figured out from most parameters’ names. But parameters spillSlotSize and flags might need further explanation.

Simply explained, spillSlotSize is the size of a register. The reason we need this parameter is that when a compiler allocates variables in a computer system, it has to decide which should be allocated to registers and which to the memory. The action of moving one or more variables from register to memory is called “spilling”. In this case, spillSlotSize is needed to determine how many memory space should be allocated. Since different kinds of registers have different sizes, parameter flags is required to identify whether they are integral registers (CPU_REGISTERS) or floating point registers (FPU_REGISTERS).

```java
public class Register(int number, int encoding, int spillSlotSize, 
                      String name, RegisterFlag... flags) { 
    this.number = number; 
    this.name = name; 
    this.spillSlotSize = spillSlotSize; 
    this.flags = createMask(flags); 
    this.encoding = encoding; 

    values = new CiRegisterValue[CiKind.VALUES.length]; 
    for (CiKind kind : CiKind.VALUES) { 
        values[kind.ordinal()] = new CiRegisterValue(kind, this); 
    }
}
```

Figure 4-2 Register definition class
Without being bothered by any details, in Figure 4-3, registers are grouped into CPU (integral) registers, FP (floating point) registers and ALL registers.

The array CPU_REGISTERS represents integral registers, which not only contains general purpose registers (r0 – r30) but also stack pointer (SP), zero register (ZR) and a mysterious r31 which in fact does not exist.

In ARMv8, SP and ZR cannot be used as general purpose registers. The stack pointer has been introduced in section 2.4.2. Zero register is used to get zero when it is needed. Actually, SP and ZR are the same register. When the instruction indicates that SP should be used, ARMv8 processor will provide the real content to the instruction. When the instruction wants to generate zero using ZR, the processor will hide the content and give zero to the instruction. There is no ambiguity of SP and ZR’s usage since they can only be accessed in different contexts.

Moreover, r31 is not a real general purpose register. It represents either SP or ZR depending on the instruction. The reason it is put here is that the numbering of the registers must be sequential. It means that we cannot skip the 32th register (r31) because the arrays need to calculate the index of all register using their sequential numbers. In our register allocation algorithms, r31 will never be used. So this workaround is safe.

The array FPU_REGISTERS is used to maintain floating point/SIMD registers (v0 – v31). These registers are used to perform large data computation or multimedia operation.
The array `allRegisters` is a simple combination of the above two arrays which is used to give the `Platform` class information to create the global register table.

### 4.3 ARMv8 Representation Evaluation

To evaluate the ARMv8 representation, we use Eclipse and Maxine's built-in python script to debug the boot process and check if the platform configuration is correct.

Figure 4-4 illustrates the creation method of Maxine VM at runtime. Within this method, the a `VMConfiguration` object is created which means the `platform()` static method successfully identifies and verifies ARMv8 platform representation and register configurations discussed in the previous sections.

```java
public MaxineVM create() {
    VMConfiguration config = new VMConfiguration(buildLevel.getValue(), platform(),
    vm(referenceScheme),
    vm(layoutScheme),
    vm(heapScheme),
    vm(monitorScheme),
    vm(runScheme));

    MaxineVM vm = new MaxineVM(config);
    MaxineVM.set(vm);
    config.gatherBootImagePackages();
    config.loadAndInstantiateSchemes(null);
    return vm;
}
```

**Figure 4-4 Creating VM**

### 4.4 Next Step

After the ARMv8 implementation is done, the next step will be implement Stubs for ARMv8.

Each instance of `Stubs` represents a block of assembly/machine code to express certain compiled parts of a Java program. `Trampolines` are stubs used to connect real instance types at runtime since some types of code can be known only at runtime. Figure 4-5 shows the structure in which `Stubs` play the role of a bridge between the Maxine VM runtime and the code generation parts.

Implementing Stubs and Trampolines involve writing basic assemblers for the target platform. In our staged plan, firstly a series of dummy assembler segments will be created as placeholder. Then they should be implemented in the order of being called.
5 Plan and Coming Tasks

The making of plan is actually one of the outcomes of the research methodology talked in section 3.

Apart from the report/thesis write activities, the implementing activities (see Figure 5-1) can be categorised into “Implementing ARMv8 platform”, “Porting unit test framework” and “Implementing Graal/T1X compiler on ARMv8”, which are labelled with number 4, 6 and 7 respectively. These three activities are condensed versions of the subgoals untangled in section 3.1 with related subgoals bound together. Figure 5-1 shows the project plan in a Gantt chart.

Within the coming weeks after submitting the progress report, the offline compiler should be activated. Then our focus moves to porting the unit test framework followed by implementing Graal and T1X on ARMv8. After that comes the final evaluation of the project.

As each activity is being performed, the activity record and implementation documents are written as concomitants of these activities. Thus, the final thesis writing activity can be done in a trackable manner.
6 References


