Comparing Alternatives to DalvikVM Bytecode Format with LLVM's Intermediate Representation

Yannick Verschueren
Vrije Universiteit Brussel (Rolnr.: 0501172)

Jennifer B. Sartor
Vrije Universiteit Brussel

This paper describes the insights and knowledge acquired during the preparation of my *Bachelorproef.* The experiment consists of accelerating code that runs on mobile devices to gain efficiency and reduce battery usage. The main goal of this project is to port an existing framework to work with Dalvik bytecode and more specifically with Android applications, which are written in Java. The framework finds code that could be accelerated using specialized hardware.
We first tried to work with the Android's Dalvik bytecode, as this was readily available by decompiling standard Android applications. Several decompilers exist, but they all have the same common problem: the bytecode they output is not in SSA form. The existing framework only accepts code that is in SSA form to extract code sequences. This paper will thus show the investigation towards a more useful representation and compare it to the LLVM representation. This comparison will be used in the second part of the project in which the code of the framework will be adapted. This adaptation of the framework will be used to accelerate Android applications, something that is not possible in the current implementation of the framework.

I. INTRODUCTION

Mobile phones and tablets are becoming a viable alternative to portable computers because of the increasing capacity and computation power of these devices. To keep mobile applications as efficient as possible, it is necessary to optimize applications and more specifically their source code as much as possible. Several techniques exist to optimize code for efficiency and memory usage, however most of these techniques require no alteration of the underlying hardware. Other projects have used specialized hardware to execute specialized instructions in order to achieve efficiency gain
This thus arises the question whether it is possible to adapt mobile applications in a way that they would benefit from specialized computational units that have custom instruction sets. Such optimizations exist [14] for applications written in the C programming language and use a modified compiler to identify common code sequences in a certain application domain. These sequences are then transformed to a custom instruction that could be implemented and executed in a new Specialized functional unit in hardware. When studying the existing framework it became clear that the Intermediate Representation used has several conditions before common code sequences can be identified. The most important conditions are a register based representation and a SSA form. An IR is used so that code optimizers can be written independently of a programming language. Although this definition opposes our need for another IR, there still exist several different IRs with each their different characteristics and origination languages. For example, the LLVM IR cannot be created from Android application. The search of an IR that satisfies the forementioned conditions lead us to the discovery of Jimple and its SSA variant Shimple. Jimple is part of the Soot framework, a Java optimization framework. This framework allows us to decompile Android applications into a suitable IR in a Static Single Assignment form and construct related control-flow diagrams.
One could argue that analyzing Java bytecode directly, after converting it to a SSA form is a viable idea, since this format is not only used by all Java compilers but also by other compilers for languages such as Python, Scheme, Prolog and

Smalltalk which can be compiled to Java bytecode [1]. However, the main reason the bytecode is not used for optimization is that it is a stack-based IR and is thus not suitable as the optimization process requires a three register instructions.
This paper will introduce Jimple, the reason why we should use it over Dalvik bytecode and compare the representation to the IR of LLVM. The framework was developed in LLVM, which is a compiler and the reason it uses the IR. Android applications are compiled for the Dalvik Virtual Machine, down to bytecode, which does not meet the conditions specified above.This comparison will lead to the second part of the project, where we will use our study to adapt parts of the existing framework. The remainder of this paper will be organized as follows: Section V will explain the Dalvik bytecode and why we do not use it in our project. Section VI gives a more detailed explanation of the Jimple representation and the Soot framework. Section II explains why it is important that the IR is in SSA form. Section VII will explain in more detail the Shimple variant of Jimple. Section III gives some more insight in the LLVM IR so that it can be compared to Shimple in section VIII.

II. STATIC SINGLE ASSIGNMENT

Static Single Assignment form is a property of certain Intermediate Representations (IR). The form guarantees that every variable has exactly one definition. When a new value needs to be assigned to the variable, a new variable is created with a similar name and is used instead of the first variable. An example illustrates the advantage of this characteristic of the SSA form:

```
  Non-SSA            SSA

if(boolean)        if(boolean)
    x = 1              x1 = 1
else               else
    x = 2              x2 = 2
                   x= φ(x1,x2)
return x           return x
```

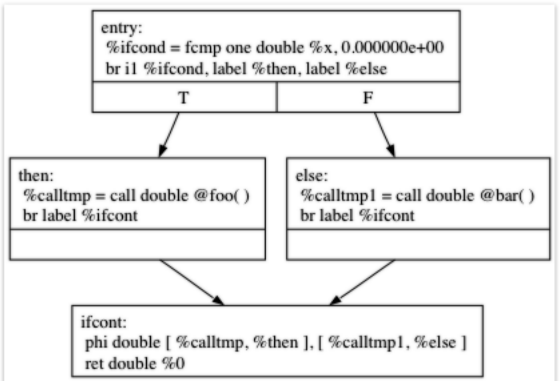
In the example we can see that the variable "x" from the non-SSA form is split into three different variables in the SSA form. This means that none of the variants of the variable in the SSA form will ever contain a different value than the one they had been defined with. It is clear that the first two statements in the SSA example will create new variables, however it is not clear what the returned value of the function is. As shown in the example this problem is solved by introducing the ϕ-function. The ϕ-function , also referred to as merge operators or choice operators, is a mechanism that allows a split variable to be merged into a single variable.[10] When we look at the ϕ-function more closely we can see that our previous definition of merging all arguments can be further refined, as a simple merge would not give any particular gain in precision. If we define the operation as a choice operator, a more interesting aspect of the operator surfaces. As we can see the operator only takes two arguments in the example. This is because it only has to choose from either the if branch or the else branch. In general a ϕ-function has as many arguments as it has preceding control flow statements. Each argument will be the value of the variable defined in its corresponding control flow branch. This increases precision of the variable, to be assigned, as it is now clearer why a particular value would be chosen.
Another example can explain the "Static" part of the Static Single Assignment form.

```
   Non-SSA               |     SSA
  x = 1                  |  x1 = 1
label1:                  |label1:
  if (x = 5) goto print  |  x2 = φ(x1, x3)
  print x                |  if(x2 = 5) goto print
  x = random()           |  print x2
  goto label1            |  x3 = random()
                         |  goto label1
print:                   |print:
  print x                |  print x2
```

We can see similar statements in this example as in the previous example, however there is a small but important difference. The variables "x2" and "x3" are defined inside a loop and will thus be assigned to several times during the execution of the loop. One should note that the print statements in the SSA form look identical but will never print an identical value. The single assignment property is violated in this example, during the dynamic execution. This is where the static keyword comes into play. If we look at the code example in a static way, it is clear that the "x2" variable is only defined once. The single assignment condition can thus be held if a variable is used in the scope of the same assignment. If an assignment is in a loop it will be executed multiple (dynamic) times, hence the word *Static* in Static Single Assignment.

A. ϕ-functions

We will now take a closer look at the ϕ-functions introduced in the previous examples. We already showed that our original definition of ϕ-functions (as a merge operator) was not sufficient when we want to explain the actual effects of the operator. To better explain the concept, we will introduce the notion of control flow graphs.
A control flow diagram consists of several nodes called "basic blocks". These blocks, consisting of several instructions or statements, should contain only one control flow instruction as the very last instruction. This instruction will create an exit point for that node. When a program is being executed normal program flow will enter a node in its entry point, the first statement of the block, and exit the basic block via the forementioned exit point. The edges in the graph represent the control flow branches between basic blocks.[11]



```
entry:
 %ifcond = fcmp one double %x, 0.000000e+00
 br i1 %ifcond, label %then, label %else

then:
 %calltmp = call double @foo( )
 br label %ifcont

else:
 %calltmp1 = call double @bar( )
 br label %ifcont

ifcont:
 phi double [ %calltmp, %then ], [ %calltmp1, %else ]
 ret double %0
```

(Image taken from llvm.org)

When the exit points of two different nodes point to a single entry point of another node, we call this node a *join* node. In most cases ϕ-functions will be placed at the beginning of these join nodes, as we can see in the example above. The example introduces us to the LLVM IR which will be discussed later in this paper. The ϕ-function has, as we mentioned before, as much arguments as it has preceding control flow operators. This is equivalent to the amount of pointers in the control flow graph that point to the join node of the ϕ-function. The function will, at runtime, be evaluated to the value of the argument corresponding to the control flow of the program. This shows our second definition of the ϕ-function, a choice operator. The example shows us that the arguments not only consist of the names of the variables from the different control flow branches, but also have a second variable that indicates for each argument its corresponding branch. The value of this variable is a choice made by the designers of the intermediate representation. In case of LLVM (shown in the example) the designers chose to use the label from the corresponding branch as an identifier for the argument. This method is used to prevent confusion, since a notation without these identifiers would be incomplete. If we define the syntax of the ϕ-function as ϕ(r1,r2,...) it would not be clear whether the function would evaluate to r1 or r2 at runtime. The study of the ϕ-functions will help us understand how they work and how it will affect the code when they are removed by the framework. It will also enable us to make comparison between ϕ-functions in LLVM and ϕ-functions in Shimple, specifically their syntax and their place in the code.

III. LLVM

Before thoroughly inspecting the Jimple and Shimple representations, we turn our focus towards the LLVM compilation framework. LLVM is a collection of low-level coding tools, such as compilers, assemblers, linkers, debuggers, etc. The name of the project that bundles these tools can be confusing, as it has no virtual machine. We will focus in this paper on its compiler component and the compiler's internal representation. Compiler designers generally use a three part design when implementing compilers. The three components are a front end and back end with an optimizer between the two main components. The front end converts code, which is inputted by the programmer, into an abstract syntax tree after checking for errors. This Abstract Syntax Tree (AST) is then used by the optimizer to execute various code optimizations. It is here that an internal representation (IR) is created, as it can be useful for some optimization operations. In our project, it sufficed to stop the compiler here and the framework is implemented using this IR . The back end eventually transforms the code into native machine code.[19]

A. LLVM in the project

We will explain briefly the framework mentioned in the introduction, as it clarifies our need to investigate LLVM and its internal representation. The framework [14] modifies the LLVM compiler and its IR code generation component, to identify valid data flow graph sub graphs (Section II). These subgraphs will be used in the creation of the Taylor expansion diagrams. The image below gives an example of LLVM code transformed into polynomials, used by the TED. The instructions in italic are binary operators and are important in the creation of polynomials. All of the instructions in the code will be explained below. LLVM has a specialized analyzing tool, called *llvm-prof*, that generates an execution profile. This execution profile is a report, in readable format, that helps to identify the *hot loops* of a program. Hot loops are regions of a program, or sequences of code statements, that are executed more often than other regions. This profile is used in the framework to focus on parts of the code that are executed more, and thus contribute more to the execution time.

```
%ip_read = call i8 @f1(i8* %ip) nounwind
%brightness_cast = zext i8 %ip_read to i24
%dpt_read = call i8 @f1(i8* %dpt) nounwind
%tmp_cast = zext i8 %dpt_read to i16
%cp_read = call i8 @f1(i8* %cp) nounwind
%tmp_1_cast = zext i8 %cp_read to i16
%tmp = mul i16 %tmp_1_cast, %tmp_cast
%tmp_2_cast1 = zext i16 %tmp to i24
%tmp_2_cast = zext i16 %tmp to i32
%area_read = call i32 @f2(i32* %area)
%tmp_4
= add nsw i32 %tmp_2_cast, %area_read
call void @f3(i32* %area, i32 %tmp_4)
%tmp_5
= mul i24 %tmp_2_cast1, %brightness_cast
%tmp_5_cast = zext i24 %tmp_5 to i32
%total_read = call i32 @f2(i32* %total)
%tmp_6
= add nsw i32 %tmp_5_cast, %total_read
call void @f3(i32* %total, i32 %tmp_6)
ret void

f1 = _ssdm_op_Read.ap_auto.i8P
f2 = _ssdm_op_Read.ap_auto.i32P
f3 = _ssdm_op_Write.ap_auto.i32P

converted polynomials

tmp_4  = area_read + cp_read*dpt_read
tmp_6  = cp_read*dpt_read*ip_read + total_read
```

The conversion of code sequences to polynomials has four steps. The first step is a read and write slicing. We check that a read instruction can be moved before and a write instruction after the arithmetic instructions. If there are dependencies, maybe only a part of the computation can be extracted. Step two is the rewriting the casted variables propagating them where they are used, castings are not important for the similarity identification in the LLVM case, because instructions are typed, see section III, and a floating point instruction is different from an integer one. The third step is the identification of the output variables.The last step is the actual creation of a polynomial with the operations that yield to each output.

B. LLVM's Intermediate Representation

Now that we have our motive to study LLVM, we will dive deeper into the internal representation. LLVM code representation resembles an abstract low-level RISC-like instruction set. However, it has several higher-level characteristics to ensure sufficient code analysis and optimization. The most important

higher-level information source is the strong typing system.
The typing system can be divided into three categories. The first category is the smallest as it only contains the void type. Void represents no value and has only limited use. The second category consists of function types. Function types are the type of value returned by a function call. The type can be either a void type, or a first class type. The first class type is the last and biggest type category. This collection can also be divided into several subcategories of types. This time four categories can be distinguished: single value types, label types, metadata types and aggregate types.
Single value types are native types that are valid in any registers. They consist of the standard types, such as integers, floats, pointers and vectors, and a special type, called x86_mmx, that is only available on x86 machines. Label types simply represent labels and require no further explanation. Metadata was introduced to ease optimization as it can be used to store information in the IR itself[20]. Aggregate types are the types of data structures that contain multiple elements of the same type. The two data structures available in LLVM are array and structures.
The second higher-level feature is the usage of unlimited virtual registers in a static single assignment form. The advantages of using SSA earlier also apply to the IR of LLVM.
As was the case in Jimple, LLVM has designed multiple variants of the IR. The first form is the textual form that is used to display code to the user and by external IR code optimizers and analyzers. Two other forms exist to be either used by internal modules or to be stored on disk in a bitcode format. The last form mainly exists to save storage space.[19]

LLVM uses only 31 opcodes to create its entire instruction set. This small amount of opcodes is the result of clever reuse of opcodes for different purposes. To start, there are no unary instructions, they are implemented by using other operations (eg. not is implemented by using a binary xor relational function). Overloading of operators is the second method and is only possible because of the typing of variables. The grammar for the IR is too large to describe in this paper, so we will analyze the IR by dividing the instructions into different categories. We will use the example of LLVM IR above to give an example of the instruction categories

First category : Terminator instructions

Terminator instructions are instructions that close a basic block. They change the control flow of a program and, therefore, create an edge when represented in a control flow graph. These instructions do not create any other value than void as only the secondary (control flow) effect is essential. The instructions consist of simple return, branch and switch statements, and more specialized functions such as invoke and indirect-branch. An indirect branch is used to jump to a basic block that is specified by a memory location, rather than a label in the case of a normal branch function. The invoke instruction is used together with unwind to implement exceptional control flow based on unwinding the execution stack. An easy way to understand the mechanism is to see the invoke function as a high-level language try statement with a second parameter that indicates the basic block that handles the exception, much like the catch statement. The unwind instruction is then simply the equivalent of the throw statement. In the example given earlier, there is only one terminator instruction: the final *ret void.*

Second category : Binary instructions

Binary instructions take, as the name suggests, two operators and produce a single result. This represents the three-register form of the IR. These operators are the most important part of the IR as they do most of the computations in a program. These are the computations is which we are most interested in this project. The return value of these functions is always the same as the type of their arguments. All binary instructions can be divided into two groups. The first group accepts only integers (signed or unsigned), while the second group only accepts floating point numbers or doubles. The base functions are: addition, subtraction, multiplication, division and the remainder functions. Each function has a version in both groups but have the same semantics. The division and remainder function can be further split into a signed and unsigned version, as division with integers can sometimes lead to unexpected results. As mentioned earlier the binary instructions in the example are in italics. For instance: %tmp = mul i16 %tmp_1_cast, %tmp_cast.

Third category : bitwise binary instructions

Bitwise binary operations use the same semantics as normal binary operators. These instructions are also important to us in this project, despite the fact that they do not occur as frequently as the binary instructions. They take two arguments of the same type and calculate some result of the same type. The main difference is the efficiency with which the operators calculate their result. This comes however with a price of reduced functionality. The instructions can be divided into two groups. The first group is used to calculate some arithmetic operation and has operators *shl* (shift left) and *lshr* (shift right). The instructions in the second group are more frequently used as logical operators but can also be used to calculate certain expressions. This group consists of *and, or and xor*. Examples of bitwise binary instructions:

- %res = shl i32 4, %var (Shifts the number 4 "var" amount of times)
- %res = and i32 25, 30 (Returns 24)

Fourth category : memory instructions

The LLVM compiler was originally created for C-like programming language so special care had to be taken to allow manual memory management. LLVM uses only two operators to access and write to memory: load and store. Because a store operation can write to multiple memory locations through the use of pointers, memory locations are not in SSA form. It would be too difficult to create a proper and compact SSA form for these memory locations. To store an element in memory, a pointer to a region of allocated memory should exist. This pointer is created by using either the *malloc* or *alloca* function. The difference between malloc and alloca is the location of the allocated memory. Memory can be deallocated by using the *free* operator. All local variables are implicitly allocated, but require no special operator to be referenced. The pointers returned by the allocation functions can be used by the load and store operations. The load operation is used to read from memory and the store operation writes a value to a given memory location. For example

```
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32* %ptr
```

will define a variable val with value 3. The *getelementptr* operator retrieves the memory address of a sub element of any given data structure. Example: (getelementptr Obj %X, i32 %i, ubyte 5) is the equivalent of X[i].a where a is the third field in the object class. Other operations include : fence, cmpxchg and atomicrmw are barely or never used in our project and will thus not be discussed.

Fifth category : casting instructions

Although casting operations are not important for similarity identification in the framework, because binary instruction are divided into integer and float operations. Nevertheless, it is important to recognize them to be able to correctly remove them. These instructions are still in three-argument form as they take a type as a third argument. All instruction take a single operand and a type and return a value equal to the given type. We will not show all the casting instructions as they are similar in functionality. We will give some examples instead.

- %tmp = mul i16 %tmp_1_cast, %tmp_cast %tmp_2_cast1 = zext i16 %tmp to i24 (snippet from the example given in part A)
- %X = trunc i32 257 to i8 (Truncation, yields 1)

- %X = fptoui double 123.4 to i32 (floating point to unsigned integer, yields 123)
- ...

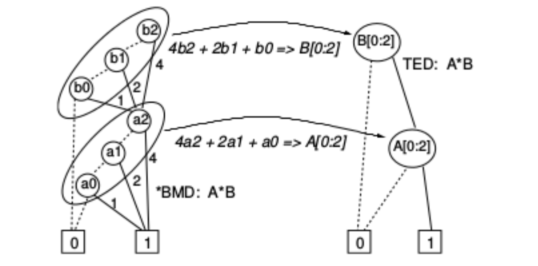
Sixth category : other instructions

The IR supports several instructions that do not fall under any of the above categories, but can be very important. Two comparison operators exist in this category: *icmp* and *fcmp*. These operators are used to compare two integers and two floating point numbers, respectively, and return a boolean value indicating whether they are equal or not. The *select* can be used as a low level ternary shorthand if function. It takes a condition (a bit) and two values of the same type. If the condition is true, it will return the first value, otherwise the second value will be the returned value of the operator.
The *call* operator is used for function calls and requires several arguments, such as return type, argument list, optional calling convention and others. The last function we will discuss is the ϕ-function. We already explained the reason of existence for this function and its semantics. All value arguments of the function should have the same type that is given as a first field in the function. Example: %x = phi i32 [%x_1, %Label1],[%x_2, %Label2],[%x_3, %Label3].
[21]

IV. TAYLOR EXPANSION DIAGRAM

We introduced the notion of a Static Single Assignment form because this is a requirement of the framework we will be using in the project. The goal of this project is to adapt the source code so that it accepts the Intermediate Representation we are studying in this paper. One of the purposes of the program is to identify common sequences of instructions throughout one or more applications. We make the distinction between sequences that are common in one application and that are common in a certain domain. When searching for common sequences it is necessary to use pattern-matching techniques that are both fast and complete. Several pattern matching techniques exist such as enumerating a subgraph of a Data Flow Graph[12], Potential Custom Instruction pattern identification and timeshaping[13] and other techniques using an acyclic Data Flow Graph. However, these techniques do not provide the best result, as they do not always find common sequences in a way that a custom instruction can be created that represent the same functionality in different forms. Using Taylor Expansion Diagrams has been proven to be more effective for finding custom instructions that can be used across applications in a certain domain.[14]

Taylor Expansion Diagrams [15] are compact, canonical representations of data flow computations. The computations can be represented by polynomials with multiple variables, closely related to the decomposition into Taylor series, on which the Diagrams are based. Taylor Expansion Diagrams were introduced because of the limitations of earlier diagrams, such as decision and Binary Moment Diagrams (BMD). In these diagrams word-level computations, or those used in all general programming language, such as "X + Y", need to be decomposed into bit-level computations before they can be represented by a graph. This composition can cause the number of variables in the diagrams to expand exponentially. This causes an increase in the memory requirement and operation time. Another method is thus used to prevent this behavior and treat the high level computations as algebraic functions rather than expanding them in bit-level components. This higher-level representation of mathematical operations is what makes the TED more useful for code matching. With the DAG representation of a different sequence code for the same functionality, unlikely to produce the same graph for the different sequences, whereas a TED will.



Abstraction of bit-level variables into algebraic symbols for : $F = A \cdot B$. *BMD: Binary Moment Diagram

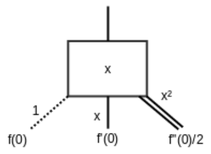
(Image used from [15])

The figure above shows how the new method can be used to represent the bit level variables into a higher level representation. The left side of the figure depicts the bit-level decomposition of the function "A*B" with respect to the bits of variable A and B. The right side shows the abstraction of these bits into a higher level symbolic representation. One can use a Taylor series expansion of the function to achieve this type of symbolic abstraction. If we rewrite the moment decomposition
$f = f_{\bar{x}} + x(f_x - f_{\bar{x}})$ as $f = f(x=0) + x\frac{\partial(f)}{\partial x}$
We can see that the right equation resembles a Taylor series expansion. We can thus represent a BMD with a Taylor series expansion by allowing the variable x to take integer values and thus discard the bit-level expansion.

The Taylor series expansion of function f(x), where $x_0 = 0$, can be represented as:

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!}(x - x_0)^k f^k(x_0)$$
$$= f(0) + xf'(0) + \frac{1}{2}x^2 f''(0) + ...$$

where f'(x) is the first derivative of f(x) and f"(x) is the second derivative. When applying this decomposition recursively to higher order functions and storing the results in an acyclic decomposition diagram, we get the Taylor Expansion diagram. The graph consists of several node sfollowing the general decomposition scheme.



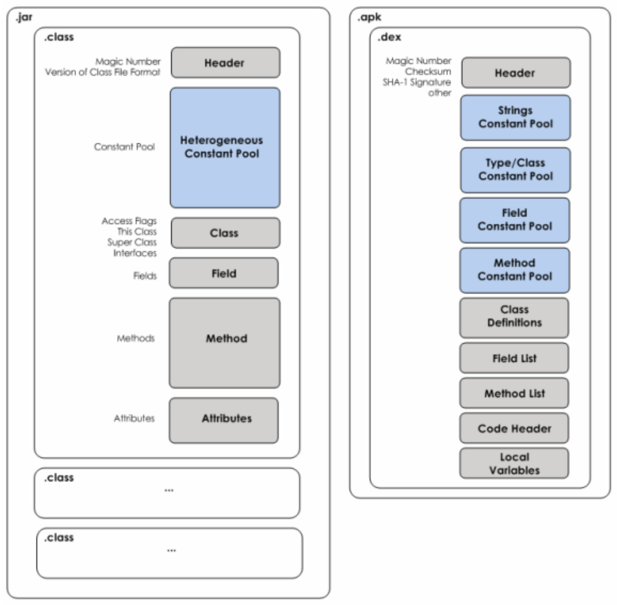
The nodes in the graph correspond to the successive derivation of the function with respect to the original variable (in case of the example: "x"). Each node thus consists of a single variable and has as many children as there are successive derivations. The first child, also called a constant Taylor expansion, has a dotted edge. The first derivation is connected via a single line, the second derivation via a double line, etc.
When we have created a TED, the next step is to reduce the size of the graph to reach the canonical representation we desire. This is done by a set of rules called, *normalization*. The general idea is to remove redundant nodes and to combine graphs of which there exist an isomorphism. A node is redundant if all of its non-dashed edges (zero edges) are connected to a terminal node zero. This means we can remove the node from the graph, redirect the incoming edges to the outgoing edges of the node without any secondary effects on the complete function. In a special case where all outgoing edges of a node point to the zero terminal node the evaluated value of the function in that node will be equal to zero. A terminal node is a node that contains only a single constant value and has not outgoing edges.
Isomorphic graphs are detected by calculating a bijection between the vertex (or node) sets of two graphs. If this bijection is such that any two vertices x and y of G1 are adjacent in G1 if and only if (u) and (v) are adjacent in G2, with f being the bijection function,G1 being the first graph and G2 being the second graph, we can talk about an isomorphism. This means that two subgraphs of a TED are isomorphic if the vertex sets and edge sets can be mapped on-to-one. They will thus represent the same function because of the definition and construction of the graphs. After the application of the rules specified above on a TED from the original mathematical expression, we end up with a normalize canonical Taylor Expansion Graph we can use to create custom expression or instruction for common code sequences.

A. Using TED for instruction clustering

To identify code sequences with similar functionalities using the normalized TEDs we have to be able to create polynomials from a series of instructions. We do this by creating a Data Flow Graph from the Intermediate Representation of the source code we want to analyze. This is where the Static Single Assignment form condition comes into play. To create a correct data flow graph for static program analysis the analyzed IR should be in SSA form[16].
When choosing a sequence of instructions for conversion to a TED, several constraints are introduced. Each list of chosen successive instructions should be a maximal convex subgraph of the original data flow graph of a given basic block. We call these graphs subDFGs. This list of instructions should thus not contain any control flow instructions, such as branch, jump, goto, etc. The subDFGs should also not contain any memory operation, such as load and store, as the custom instructions created from these sequences can only include operations executable in a functional unit. These units cannot execute memory operations as they are not connected to the memory bus. They are only connected to the processor's registers. The exploration of the DFG is done using a binary search algorithm [14] which we will not describe in this paper.
The second step is to transform the previously found subDFGs into a polynomial expression. With the polynomials we can create a TED using a Taylor Expansion series, as mentioned above. After reduction and normalization of the Diagram, the result is a unique canonical representation of the polynomial. Because of the uniqueness of this representation, one can easily verify if two polynomials have the same functionality by inspecting their TED representation. In this way similarity can be detected even if the data flow graphs are not identical. This is where other pattern matching techniques fail as they do not construct a TED.

V. DALVIK

Dalvik is Android's mobile alternative to the Java Virtual Machine. It uses a limited but similar implementation of the standard Java libraries. This means that Android applications are written in standard Java, but are compiled to Android bytecode. This bytecode is very different from the standard Java bytecode because of the various constraints the Android platform. These constraints can go from limited processor speed and memory, to low storage space and slow data transfer speeds. Because of this it is important that Android bytecode is both efficient and small in size. Virtual machines often favor a stack-based architecture over register-based architectures for reasons of code simplicity and code density. This means that executables for register-architectures are generally larger than executables for stack-based architecture [3]. However, the simplicity and density come at a performance cost. It has been shown that a register architecture requires an average of 47% fewer instruction executions than their stack based variants. However, register code is about 25% larger than corresponding stack based code. The performance gain from using a register based architecture, taking into account every advantage and disadvantage, is on average a 32.3% lower execution time for standard applications [4].
Given this performance gain from using a register-based architecture, it seems advantageous to use it for devices with limited processing power. This is why Android chose to transform stack based Java bytecode to a register based format. The increased code size is countered by Android by compressing all Java class files into a single file in the ".dex" format. This is done by using the "dx" tool after all Java source code files are compiled with the standard Java (javac) compiler. This Dex file will be loaded and executed by the Dalvik Virtual Machine. The Dex format has been optimized for both memory usage and code size. This optimization is mainly done by using a shared, type-specific constant pool. This pool stores all literal constants used by all Java classes it embodies. This includes not only string constants, but also field, variable, in-



terface, class and method names. In the code itself these constants will be replaced by the appropriate index in the constant pool rather than storing them throughout the code. This method allows for a great reduction of memory usage and code size, because of the high chance of duplication elimination [2]. The figure below gives a graphical representation of the pool sharing in Android

The Dalvik bytecode will not be used in this project as it is not in SSA form. A first idea was to transform Dalvik bytecode into an SSA form, but this process would be too difficult and would fall out of the scope of this project. The necessity for an SSA form is explained in Section II. Nevertheless, it is important to understand how Android compiles and compresses Java source code into their Dex format as it will be used to create the Jimple intermediate representation.

VI. JIMPLE

Jimple stands for Java sIMPLE [8] and offers a great alternative representation of Dalvik bytecode, as there exists an SSA version of this representation. Jimple is the internal representation used in Soot, developed by the Sable Research Group, a Java code optimizer. Jimple was created because optimizing Java stack-based bytecode introduces unnecessary complications such as: inexplicit expressions, arbitrarily large expressions and the fact that simple transformations can become complicated. When implementing the Jimple transformation one of the goals the designers wanted to reach the ability to produce correct Jimple code for any verifiable bytecode.

"Even for bytecode that does not come directly from a Java compiler" [5]

This is especially useful in our project as we want to create Jimple code directly from the Android bytecode. The intermediate representation has some essential characteristics that define it. The first characteristic is the stackless nature of the code. By now it is clear that using a register based representation is important for both performance and compatibility with the sequence-identifying framework we will be using. The second characteristic is the restrictions of expressions. Expressions are restricted to a small number of operands and they must be either constants or local variables. These local variables must have explicitly declared types. The typing of local variables was implemented to allow higher order optimization such as type based code analysis. Since Jimple is mainly used for code optimization, one last requirement for the representation is that the code is compact. This means that the produced code should have the lowest amount of statements as possible. We will now show the steps in the creation of Jimple code from Java bytecode. We will then show the transformation of Android files to Jimple, as it it different in some aspects.

A. Creating Jimple from Java bytecode

Producing Jimple is a five step process starting with Java byte code, ending up with correct, stackless, typed and compact Jimple code, thereby satisfying all earlier given conditions. We summarize the process here, because it will give us a better insight on how Jimple operates internally and the representation.

1. Step one : Producing typeless Jimple

The first step in the process is producing typeless incompact Jimple statements from the Java bytecode. This is done by transforming the operand stack to a series of Jimple variables without a type and by successively changing each reference to the stack by explicit variable references. An important concept here is that equal Java byte code statements do not necessarily transform into the same Jimple statement. This is a direct result of using a stack to pass around arguments to functions. For example: given the *iload 0* instruction; when transforming the instruction for the first item, the value of the variable 0 is assigned to the variable representing the first element of the stack. However, when the instruction is transformed more than once, successive transformation should not reassign the first element of the stack. This means that the transformation has to keep track of the height of the stack. Calculation of the height of the stack is then done by traversing the byte code and using a set of rules to correctly increase or decrease the stack height. This set of rules is too extensive to give here as it has an entry for every single byte code instruction. An interesting note to make is that this first step in the process is enough to create correct Jimple statements. This representation could be used for optimization and code analysis, albeit not very efficiently.

2. Step two: Compacting

The first step creates code that is not compact enough for efficient operation and analysis purposes. The second step is thus compacting the produced code. Compacting is done by using *copy-constant propagation* and simple dead-code elimination. The algorithm used during the transformation uses data-flow analysis to assure correct code production. Propagation and dead code elimination is executed as long as the code does not change, providing the maximal level of possible compaction without changing the functionality of the code. Several experiments have been conducted in which the location of the second (compaction) step of the process was moved after typing of the Jimple statements. When the compaction is done after typing of the statements, a decrease in code size is visible. However, when small code size is not a primary objective, the compaction should remain the second step in the transformation as it gains a boost in execution speed[5]. As is in the first step, we could now stop the process, as we have created correct Jimple statements.

3. Step three : Splitting local variables

The third step is a preparation step towards the typing of all the variables and methods, which is step four. We can not type the statements created by the second step of the process, because a variable can be used for different types of values. This is caused by reusing untyped variables in the first step of the process. The Java VM allows such behaviour of both local variables and stack pointers, provided that no variables have conflicting types. The preparation is done by splitting all local variables that are used for multiple types of values. This is done by simple renaming of the variable and corresponding arguments of function calls. This closely resembles the renaming step in the creation of an SSA form, which we have explained earlier.

4. Step four : Typing

The third step enabled us to do the actual typing of the compact Jimple statements. Typing local variables is done by inspecting the instructions that use or manipulate the variables. However since the Java Virtual Machine specifies that every variable has to be defined or declared before it can be used in any other statement [22], Jimple designers chose to use only definitions for inferring types of variables. This is trivial for primitive types, such as doubles, float, string, etc. Local variables that are assigned to an instantiation of (user-defined) classes, prove to be more difficult. The algorithm used to derive the correct type will not be given here, although the main idea is to find a common superclass of two different classes, and typing all variables to this superclass. Both methods specified above work in most general cases and only fail under certain conditions. Null pointers prove difficult as they are converted to an object type. This can produce type errors instead of the expected null pointer exception. The second problem comes from interfaces. Interfaces can introduce type conflicts in the algorithm for finding a common superclass.

The choice of using only definitions for type inference is the main source of the problems specified. A new algorithm was created by E. Gagnon that is proven to correctly find the types of 100% of the tested methods. The algorithm uses a program transformation to type methods that cannot be type inferred by the traditional methods[18]. A new version of Jimple exists in which this new algorithm is used, and does not have the problems specified above.

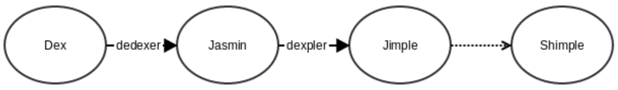
5. *Step five : Packing local variables*

The last step in the process is a second compaction step. By splitting local variables in step three of the process, some unpacking has occurred. Compact code means that both the number of statements and the number of variables should remain as low as possible. We can see this step as the reverse process of creating an SSA form, as the goal of this step is to reuse as many variables with the same type as possible. When the last step of the process is completed, typed and compact Jimple code will have been created.[5]

B. Transforming Dex files to Jimple

We use the Soot framework to transform Androids Dex files to sequences of Jimple statements. These instructions then eventually get transformed into Shimple statements which we will use later in the project. The conversion from Dex to Jimple is not trivial and requires several different steps. The figure below shows an overview of the steps.

Most developers do not publish the source code of their applications, written in Java, so one must analyze the compiled bytecode instead.



Soot was extended with a module called Dexpler, a Dalvik to Jimple converter tool. The dexpler tool uses an existing Dalvik disassembler called Dedexer [6] to create a Jasmin [7] like ASCII description of the Dalvik instructions, which will be used in the actual process of creating Jimple statements.

The process of creating Jimple statements from Dalvik bytecode is done through several steps and is very similar to the process of creating them directly from Java code, which we have described in this paper. The first step of the process is the mapping of Dalvik instructions to Jimple statements. The Dalvik opcode list consists of 276 entries [23], These contain, however, odex instructions and several instructions that are generally never used in Android applications. Odex instructions are never created by the Dalvik compiler, they are generated inside the Dalvik Virtual Machine to optimize the bytecode. Still, the number of (used) Dalvik opcodes is much greater than the available Jimple statements. One of the reasons the mapping is viable is because of the absence of typed constants and registers in the Dalvik bytecode. Because of a lack of types, several opcodes exist for the same type of operation, each with a different type (eg.: add-int,add-long,add-float,add-double). The opcodes can be combined into one Jimple statement, as Jimple uses typed variables and arguments. Using this combination method, we can divide the instructions into five large groups: the move instructions, branch instructions, method invoke instructions, logic instructions and arithmetic instructions. Each group will consist of several Dalvik instructions and fewer corresponding Jimple statements. The second step is more tricky as it involves deducing the type of arguments and variables using the Soot fast typing component. In most cases this will cause no problem because the type can simply be read from the opcode as we illustrated above. However some instructions do not provide enough information to deduce the correct type and will cause an error in the fast typing process.An example of an instruction is "const vAA, #+BBBBBBBB". This instruction moves a given literal value into the specified register, but there is no way to deduce the type of this value. Another possible problem is the null initialization:

```
int x = 0;         |00: const/4 v0,#int 0
Object obj = null;|01: const/4 v1,#int 0
                   |
(Java)             |(Dalvik)
```

As we can see in the Dalvik code there is no visible way to distinguish a null pointer from the integer zero value. Dexpler handles this problem by leaving the type open until it is used in further instructions of which the type of the operands is known. The algorithm, described by R. Milner [9] is used to extract the type of variables by using a depth-first search in the control flow graph created internally by the Soot framework.[8]. A third and last step is an optimization step, that removes some unnecessary or redundant instructions from the generated code.

VII. SHIMPLE

Shimple is the SSA variant of the Jimple representation we discussed earlier. Creating the Shimple statements is done by transforming Jimple statements into an SSA form.

A. Creating Shimple statements

Transforming Jimple to Shimple is done with algorithms that are closely related to general SSA transforming algorithms. The general mechanism of generating SSA code is the introduction of ϕ-functions and the renaming of variables. The insertion of ϕ-functions is non-trivial and will be further examined. The algorithm used by Shimple was formulated by Cytron et al. and constructs a minimal SSA form [11]. The location of the inserted ϕ-functions is the first problem that is solved by the algorithm. The minimal SSA form has two conditions that must hold:

1. In a join node of a control flow graph, which has two or more incoming control flow edges, a ϕ-function should be introduced as the first statement of the basic block. The function should have as many arguments as there are incoming branches.

2. The insertion of ϕ-functions should be limited, so that the number of inserted functions is minimized.

The ϕ-functions introduce a new assignment, so it possible that an introduction of the function may require another operator to be inserted in a node further in the control flow graph. Using these conditions, it is however not assured that the number of introduced ϕ-functions is the minimal amount necessary for correct functionality of the program. The first condition can cause introduction of ineffectual, so called *dead*, ϕ-functions. The reason that Shimple does not handle these exceptions, is because the dead ϕ-functions can sometimes be used for program equivalence or other analysis purposes. Shimple is also very rarely used for actual program assembly and will be often transformed back into Java bytecode when deploying the actual program. Transforming SSA form back to normal (non SSA-form) will then remove these unwanted ϕ-functions. Using the two conditions, the exact locations, for inserting the ϕ-functions, can be found by using a *dominance frontier*.

The second step towards creating the SSA form is the renaming of the variables. As we saw earlier, one of the properties of SSA is the unique name for every variable that is defined or assigned. Renaming the variables is done by using a top-down tree traversal beginning at the first node of the dominator tree, constructed during the introduction of the ϕ-functions. An array, that contains a single integer for every variable in the code, is used. The integer is initialized with zero for every variable and is increased whenever the variable is reassigned. This increment of numbers is then used to create unique suffixes for the new variable names. Another array, containing a stack of integers, keeps track of the scope of the definition or assignment of a variable. This stack is used to rename the new variable correctly using the other suffix array. These

stacks are used because the algorithm works in depth-first manner. This means assignments in one branch of the dominator tree could kill the definition of a variable renamed in a earlier visited branch.
As Jimple is created from Java code, we have to take exceptions into account. Exceptions have the ability to control the flow of a program in an unexpected way if we do not introduce the correct ϕ-functions. As Shimple uses Soot's internal control flow graphs to create ϕ-functions, the responsibility of correctly representing exceptional control flow lays entirely in the hands of Soot design. One of Soots control flow graph implementation, called *CompleteBlockGraph*, can be used for correct handling of the exceptional control flow, as it uses the most strict condition for exceptions. The implementation assumes that any statement inside a try block can throw an exception. The problem with this representation is possible size of a try block. Java programmers tend to keep try block as small as possible, but a code generator does not follow this general coding rule. Java has a limit on the size of a method, namely 65535 bytes[24]). When using the CompleteBlockGraph every statement in a try block should have its own argument in the ϕ-function of the corresponding catch block. Another issue introduces itself when the try block is manipulated, the dominance frontier uses a bottom-up technique so this is possible. Each introduction or removal of a statement in the try block would mean a change in the amount of variables in the ϕ-function of the catch block. Lastly, removal of the ϕ-function when converting back into non-SSA form would introduce as many assignments as there are arguments in the function. As we mentioned before, Shimple is not often used outside of code analysis, so transformation into normal form is not an infrequent operation. The solution used by Shimple's designers is to remove repeating arguments in the ϕ-function. Not all statements in a try block define or assign a variable, but are still taken into account. This causes a repetition of arguments between statements that do not alter the variable. The repeating arguments are simply removed and their control flow is added to the dominating argument, the argument pointing to the last definition or assignment. This problem only exists because of the Soot developers choice of control flow graph. The developers had no intention of creating an SSA form of their representation and thus did not take into account the problem mentioned above. Several efforts were made to create a new control flow representation, however the resulting tree does not provide any benefit over using the ϕ-function trimming[17]. This can affect us in this project, as a ϕ-function with too many arguments would require us to do the trimming ourselves.

B. Shimples Intermediate Representation

The Intermediate Representation generated from the Shimple module is very similar to the Jimple representation. This is because it uses the latter to create its statements. When describing the representation it thus suffices to analyze Jimple's IR. Soot uses two different versions of the representation, external and internal. The external representation is the regular text file which represents source code in Jimple form. This version is only used to display the result of the transformation, and to store the representation for other compilers or code optimization platforms. Using text files internally is, however, not very efficient, as manipulating ASCII text files to perform optimizations is often inconvenient. Jimple thus also provides an internal representation in the form of an API written in Java. We will not describe this API as it is of no use to us in the context of this project. We will only be using the external representation.

Jimple has only fifteen statements on which the whole grammar is based. We give an excerpt of the complete grammar the includes these statements.

```
stmt = breakpoint_stmt | assign_stmt |
enter_monitor_stmt |
goto_stmt | if_stmt | invoke_stmt |
lookup_switch_stmt |
nop_stmt | ret_stmt |
return_stmt | return_void_stmt |
table_switch_stmt |
throw_stmt;

breakpoint_stmt = "breakpoint";
```

```
assign_stmt = variable "=" rvalue;
identity_stmt = local ":=" identity_value;
enter_monitor_stmt = "entermonitor" immediate;
exit_monitor_stmt = "exitmonitor" immediate;
goto_stmt = "goto" label;
if_stmt = "if" condition "then" label;
invoke_stmt = invoke_expr;
lookup_switch_stmt =
"lookupswitch(" immediate ")
{" cases " default: goto " label "}"
nop_stmt = "nop";
ret_stmt = "ret" local;
return_stmt = "return" immediate;
return_void_stmt = "return" ;
table_switch_stmt =
"tableswitch(" immediate ")
{" cases " default: goto " label "}"
throw_stmt = "throw" immediate;
```

The limited number of statements are enough to cover the more than 200 available statements in Java. The table_switch and lookup_switch instruction correspond to the JVM eponymous statements. These statements implement the Java switch case in two different ways. Table switch uses a jump table to find the target address in an O(1) operation. The lookup switch uses labels to identify the correct address and is thus a O(log(n)) operation as label should always be sorted and a binary search algorithm is used. The assign statement for the Shimple representation is slightly different as it has to assure the single statement property of the SSA form. When assigning an already existing variable, a generated suffix should be appended to the variable if necessary. Lastly, the Shimple representation also contains the earlier discussed ϕ-functions. These functions do not exists in Jimple and are thus not present in the grammar.

The next excerpt illustrates the typed nature of the representation

```
label = identifier;
local = identifier;
constant = double_constant |
float_constant |int_constant|
long_constant |string_constant|
null_constant;
type = int_type | long_type | float_type|
double_type | ref_type |
stmt_address_type | void_type;
exception_range =
".catch" ref_type "from" label
"to" label "using" label;

method_signature = identifier
                    "(" type_list "):" type;
field_signature = identifier ":" type;
```

The grammar teaches us that every variable is typed, with the available types being:

1. double 2. float
3. integer 4. long
5. string 6. null

Every method in the representation and its arguments are typed with one of the above specified types. Labels are presented as identifiers, this means that labels can not be used as values for variables and passed around as arguments for functions. This is a feature that is often implemented in lower-level languages such as x86-assembly [25]. We do not give the remaining of the grammar for the Jimple representation as it would not contribute to a better understanding of the IR. It will be used however when comparing Shimple and Jimple's representation to LLVM's IR which is in section VIII.

C. Examples of Shimple code

We will now give some examples of generated Shimple and Jimple code that illustrate all characteristics and properties of both Jimple and Shimple we have discussed.
Java source code was extracted from the 0xbench benchmark suite.

```
final static int SEED = 113;

public double integrate(int Num_samples){

 Random R = new Random(SEED);

 int under_curve = 0;
 for(int count=0;count<Num_samples;count++){
    double x= R.nextDouble();
    double y= R.nextDouble();
     if ( x*x + y*y <= 1.0)
         under_curve ++;
    }
 return ((double)under_curve/Num_samples)*4;
}
```

The code snippet above is used in of the benchmark functions from the suite [26]. The code comes from the SciMark section featuring the integration function.We will use the suite for this project as it contains a lot of mathematical operation. This will ease the creation of polynomials. We first give the corresponding Jimple code as this is the first representation that is generated by the Soot framework.

```
public double integrate(int)
   {
       int $i0, $i1, $i2;
       jnt.scimark2.Random $r0;
       double $d0, $d1, $d2;
       byte $b3;
       $i0 := @parameter0: int;
       $r0 = new jnt.scimark2.Random;
       specialinvoke $r0.<f1>(113);
       $i2 = 0;
       $i1 = 0;
    label1:
       if $i1 >= $i0 goto label3;
       $d0 = virtualinvoke $r0.<f2>();
       $d1 = virtualinvoke $r0.<.f2)>();
       $d2 = $d0 * $d0;
       $d0 = $d1 * $d1;
       $d2 = $d2 + $d0;
       $b3 = $d2 cmpg 1.0;
       if $b3 > 0 goto label2;
       $i2 = $i2 + 1;
    label2:
       $i1 = $i1 + 1;
       goto label1;
    label3:
       $d2 = (double) $i2;
       $d1 = (double) $i0;
       $d2 = $d2 / $d1;
       $d2 = $d2 * 4.0;
       return $d2;
   }
Functions names are long and
not relevant, they are abbreviated.
```

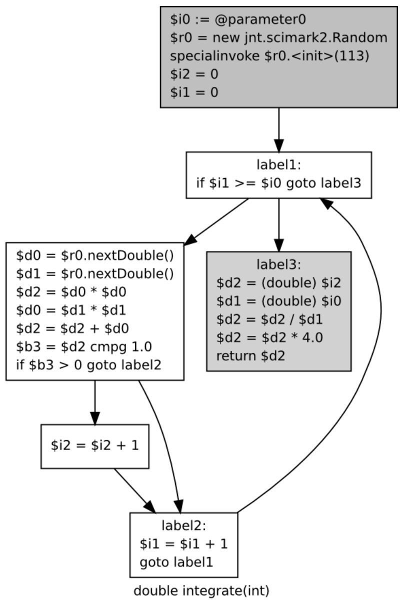
Jimple code can be seen as a hybrid between high-level Java source code and a lower level byte-code representation. We still have a notion of function definition, without the special *.method* keyword, although lower-level language concepts are clearly present, such as label and goto statements. All variables used in the method are declared in the beginning of the method definition. All variables are typed with either built-in primitive types or user-defined types (Random in the example). Variables can be cast to a different type by using a traditional higher-level method. The variable name is preceded by the desired type between brackets. When looking at the assignment operators, we see a subtle but important difference between normal assignment (=) and identity (:=). The difference between the two statements becomes clear by further inspection the grammar.

```
assign_stmt = variable "=" rvalue;
identity_stmt = local ":=" identity_value;
identity_value = caught_exception_ref |
                   parameter_ref | this_ref;
variable = array_ref | instance_field_ref |
                   static_field_ref | local;
```

An assignment can take any variable or reference, to an array or an object, and a rvalue, much like in any other programming language. The identity can however only bind a local variable to an identity value: an exception, parameter or *this* pointer. An interesting phenomenon is the introduction of a byte typed variable to represent a boolean value, instead of reusing an existing integer variable.

We now look at the Shimple representation for the given Java function above. As Shimple is created from the Jimple statements, it will be very similar to the code given above. The first thing to notice is the sizable increase in code length. This is due to the introduction of new variables and ϕ-functions. When looking at the first variable declaring line, five new variables can be identified. All the new variables consist of a base variable name with a unique suffix appended. This is because Shimple is in SSA form. Second we identify the ϕ-functions and their corresponding control-flow branch indication. In the Shimple representation, this indicator consists of a digit indicating the last statement of a basic block. The ϕ-functions choose the correct variable by using the correct digit,prefixed by "#".

```
public static final double integrate(int)
    {
        int $i0, $i1, $i2, $i1_1, $i2_1,
             $i2_2, $i2_3, $i1_2;
        jnt.scimark2.Random $r0;
        double $d0, $d1, $d2, $d0_1, $d1_1,
              $d2_1, $d0_2, $d2_2, $d2_3,
            $d1_2, $d2_4, $d2_5;
        byte $b3;
        $i0 := @parameter0: int;
        $r0 = new jnt.scimark2.Random;
        specialinvoke $r0.<...>(113);
        $i2 = 0;
(0)     $i1 = 0;
    label1:
        $d0 = Phi($d0 #0, $d0_2 #3);
        $d1 = Phi($d1 #0, $d1_1 #3);
        $i1_1 = Phi($i1 #0, $i1_2 #3);
        $i2_1 = Phi($i2 #0, $i2_3 #3);
        $d2 = Phi($d2 #0, $d2_2 #3);
        if $i1_1 >= $i0 goto label3;
        $d0_1 = virtualinvoke $r0.<...>();
        $d1_1 = virtualinvoke $r0.<...>();
        $d2_1 = $d0_1 * $d0_1;
        $d0_2 = $d1_1 * $d1_1;
        $d2_2 = $d2_1 + $d0_2;
        $b3 = $d2_2 cmpg 1.0;
(1)     if $b3 > 0 goto label2;
(2)     $i2_2 = $i2_1 + 1;
    label2:
        $i2_3 = Phi($i2_1 #1, $i2_2 #2);
        $i1_2 = $i1_1 + 1;
(3)     goto label1;
    label3:
        $d2_3 = (double) $i2_1;
        $d1_2 = (double) $i0;
        $d2_4 = $d2_3 / $d1_2;
        $d2_5 = $d2_4 * 4.0;
        return $d2_5;
    }
```



```
$i0 := @parameter0
$r0 = new jnt.scimark2.Random
specialinvoke $r0.<init>(113)
$i2 = 0
$i1 = 0
```

```
label1:
if $i1 >= $i0 goto label3
```

```
$d0 = $r0.nextDouble()
$d1 = $r0.nextDouble()
$d2 = $d0 * $d0
$d0 = $d1 * $d1
$d2 = $d2 + $d0
$b3 = $d2 cmpg 1.0
if $b3 > 0 goto label2
```

```
label3:
$d2 = (double) $i2
$d1 = (double) $i0
$d2 = $d2 / $d1
$d2 = $d2 * 4.0
return $d2
```

```
$i2 = $i2 + 1
```

```
label2:
$i1 = $i1 + 1
goto label1
```

We have unfortunately no proper way of generating a control flow graph of the Shimple code with Soot. We can however use the CFG of the Jimple code to illustrate the ϕ-function and their indicators. Only the basic block corresponding to label 1 and label 2 contain a ϕ-function. The number of arguments for the function is equal to the number of incoming arrows of the blocks, as we can see on the the figure (two in this case).

VIII. COMPARING SHIMPLE TO LLVM

After having explored Shimple, Jimple and LLVM's IR in detail, we will compare Shimple and LLVM. This comparison will be used in the second part of the project. This list of differences and similarities will be used to change the code of the framework we will adapt. This overview will often refer to different sections of the paper. The list will only feature items that are not immediately visible from the study of the representations above, such as using a dollar sign instead of a percent sign before name variables, type names being different (i32 vs integer), ending statements with a semicolon, etc.

A. Similarities

SSA Both representations are in a Static Single Assignment from. This form and the reason why it is important for these representations is studied in section II.

3-register instructions A three-register notation means that most instructions, with some exceptions, take two arguments and produce a single result. This is as opposed to a stack based representation, in which instructions only have one operand and produce one single result.

Phi functions The need for ϕ-function is a direct result of being in an SSA form. The function are used in the control flow and are a key element of the SSA form. The semantics and working of the functions are explained in section II.

Virtual registers Both IRs use virtual registers. This means that the amount of variables used in the code is not limited to the physical amount of registers in the processor.

Conversion of bitwise operations A part of the conversion of code sequences to polynomials is the conversion of logical and bitwise operators to arithmetic operations. However since both IRs represent boolean values as a value of either zero or one, it is trivial to transform these operations. For example: A and B can be converted to A * B. The creation of polynomials is explained in section IV.

Typing Both representations use strong typing with a direct result of a reduced amount of instructions. By using typed variables, instructions can be overloaded and thus reused for several types. LLVM distinguishes between decimal numbers and integers, while Jimple uses the same operators for every type.

B. Differences

Phi functions Although both representations use ϕ-functions, there is a discrete but nevertheless important difference in the usage of the function between Jimple and LLVM. The variable, representing a basic block, that indicates which value will be chosen is different. In the case of LLVM the label that starts the basic block is used, while Shimple chose to use a special digit. In the textual representation of Shimple a digit is placed before the last statement of a basic block. This number is the used in the ϕ-function as the indicator

Casting Because of the typing of variables, both IRs had to introduce a way of casting variable to a different type. LLVM uses a special instruction to perform different kinds of type casting. We discussed this in section III. Jimple, however, casts a variable by inserting the type in brackets before the name of the variable.

Function calls Function calls differ not only in syntax, but also use different keywords to perform the function call. The syntax differs in the fact that LLVM requires a return type before the call statement, while Jimple has no such requirement and uses the return type of the definition of the function. The keyword used by both representations is different as *invoke* and *call* are not the same in LLVM. Shimple only uses invoke for function calls (this can be seen in the grammar in section VII) and throw for exceptions. LLVM on the other hand uses the call keyword for function calling, and invoke to throw exceptions (it has no throw statement).

Binary operators Binary operators use a different notation between LLVM and Jimple. LLVM uses a prefix notation that is comparable to lower level languages such as assembly. Jimple uses a more traditional infix notation. This notation is possible due the fact that binary operators can be used for every type of variable in Jimple.

Memory access Because LLVM's compiler is mainly used for compilation of C programs, it has special memory access operators, such as load and store. Jimple was created for representing Java code, which has no manual memory management, and thus has no explicit memory access operators available for the programmer. All memory access functions are handled by the JavaVM.

IX. CONCLUSION

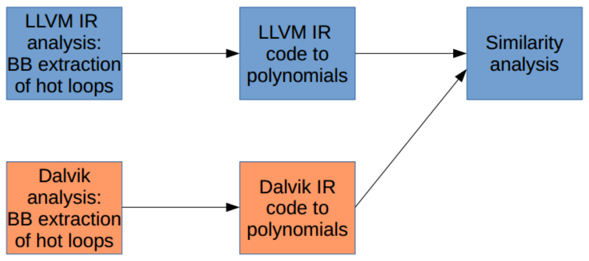
Our aim in this paper has been to find out how to represent DalvikVM bytecode in a way it could be used to identify code sequences and create polynomials. A highly specialized framework designed to work with LLVM's Internal Representation was already created to perform the identification and creation steps. Our first task was to study the Dalvik bytecode, to understand how we could use the framework with Android applications and what had to change. In this study we found out that one of the conditions for the input was not met by the Dalvik bytecode. A Static Single assignment form of the representation of the input code is required to create correct data-flow graphs. These graphs are used by a part of the framework to identify usable code sequences and to create polynomials. Dalvik bytecode is not in SSA form and is for that reason of no use in our project. The search for a better representation began, and led to a Java optimization framework, called Soot. They had created an internal representation of their own and had developed an SSA form of this representation. Fortunately they implemented a Dalvik translator module, to create their IR directly from Android code files. Finding this IR meant we had to compare it to the LLVM IR as it was the representation of choice for the existing framework. We explored both representations and their characteristics, to be able to compare them. Both representations work in a similar manner in the general sense, and for the most part only differentiate in syntax and small details. The main features we need in the project are mostly similar, such as being in SSA and using a register based architecture.

X. CONTINUATION OF *BACHELORPROEF*

This study of internal representations will be used in the second part of the project. Studying both IRs not only gave us deeper insights in the larger domain of code optimization, but enables us to understand the choices made by designers of internal representations. These choices are what led us to the discovery of a new IR that we can use in this project.

In the second part of the project, we will use our study to port the existing framework to work with Java-esque applications.

The image above shows the steps that should be taken in the second part of the project. Blue elements of the image are modules of the frame-


LLVM IR analysis: BB extraction of hot loops
LLVM IR code to polynomials
Similarity analysis
Dalvik analysis: BB extraction of hot loops
Dalvik IR code to polynomials

work that are already implemented, but are written to work with LLVM. The orange elements represent the Dalvik/Shimple equivalent that needs to be implemented. The comparison we made in this paper will be used to adapt and reuse code of the current implementation. The conversion of the different modules of the framework should not be hard as our study has shown that there are a lot of similarities between the two IRs. The similarity of the two IRs and the broad knowledge we now possess of the two representations will help us greatly in the identification of the pieces of code that will need to change and what their functionality is. Analyzing the Taylor Expansion Diagrams gave us an idea of what the output of the framework should be.

This project will look towards the future of mobile technology by making mobile applications as efficient as possible. The creation of specialized hardware that can be used for a range of applications in a certain domain allows for an increase of performance, while reducing energy usage. Our project will estimate how much Android applications would benefit from this specialized hardware and will provide the tools to create the complex instructions used in this hardware.

[1] Robert Tolksdorf, *Programming languages for the Java Virtual Machine JVM and JavaScript.* `http://www.is-research.de/info/vmlanguages` 2006.

[2] David Ehringer, *The Dalvik Virtual Machine architecture* 2010

[3] Jones Derek, *Register vs. stack based VMs.* `http://shape-of-code.coding-guidelines.com/2009/09/17/register-vs-stack-based-vms` 2009.

[4] Security Engineering Research Group, *Analysis of Dalvik Virtual Machine and Class Path Library* 2009

[5] Raja Valle,Laurie J. Hendren, *Jimple: Simplifying Java Bytecode for Analysis and Transformations*

[6] *Dedexer documentation* `http://dedexer.sourceforge.net` 2011

[7] *Jasmine documentation* `http://jasmin.sourceforge.net/` 2005

[8] Alexandre Bartel,Jacques Klein,Yves Le Traon *Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot* 2012

[9] Robin Milner, *A Theory of Type Polymorphism in Programming* `https://courses.engr.illinois.edu/cs421/sp2013/project/milner-polymorphism.pdf` 1978

[10] Allen Leung, Lal George *Static Single Assignment Form for Machine Code* 1999

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen,Mark N. Wegman *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph* 1991

[12] Kubilay Atasu, Oskar Mencer, Wayne Luk, Can Ozturan *Fast Custom Instruction Identification by Convex Subgraph Enumeration* 2008

[13] Nagaraju Pothineni, Anshul Kumar, Kolin Paul *Application Specific Datapath Extension with Distributed I/O Functional Units* 2007

[14] Cecilia Gonzalez-Alvarez, Jennifer B. Sartor, Carlos Alvarez, Daniel Jimenez-Gonzalez, Lieven Eeckhout *Accelerating an Application Domain with Specialized Functional Units* 2013

[15] Maciej Ciesielski, Priyank Kalla, Serkan Askar *Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs* 2006

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen,Mark N. Wegman, F. Kenneth Zadeckt *An Efficient Method of Computing Static Single Assignment Form* 1998

[17] Navindra Umanee *Shimple: An investigation of Static Single Assignment* 2006

[18] Etienne M. Gagnon, Laurie J. Hendren, Guillaume Marceau *Efficient Inference of Static Types for Java Bytecode* 2000

[19] Chris Lattner *The Architecture of Open Source Applications : LLVM* 2014

[20] `http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html`

[21] `http://llvm.org/docs/LangRef.html`

[22] `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`

[23] https://developer.android.com/reference/dalvik/bytecode/O

[24] `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.7.3`

[25] `http://www.aldeid.com/wiki/X86-assembly/Instructions/lea`

[26] https://code.google.com/p/0xbench/