Time Predictable Dalvik v0.1

An analysis of the DVM opcodes with focus on timing predictability

Project Report

dpt105f15

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

AALBORG UNIVERSITET

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Student Report

Title:
Time Predictable Dalvik v0.1

Theme:
Programming Technologies (PT)

Project Period:
Spring Semester 2015

Project Group:
dpt105f15

Participant(s):
René K. Hornbjerg

Supervisor(s):
Bent Thomsen

Copies: 3

Page Numbers: 30

Date of Completion:
May 9, 2015

Abstract:

This project presents an analysis of the opcode implementations within the Dalvik Virtual Machine (DVM). The analysis targets the portable version of the iterative interpreter with focus on timing predictability. A way to refactor each implementation follows its inspection.
Furthermore, the project includes a discussion on additional work needed to make the DVM fully timing predictable. The discussion points out the need for a different programming model as well as the need for a replacement or removal of the underlying Linux kernel.

Preface

This report is written by a single software-engineering student attending 10^{th} semester at Aalborg University. The semester project was started on the 1^{st} of February 2015, and finished on the 9^{th} of May 2015.

This report will concentrate on subjects related to computer science. Therefore, it is assumed that the reader has equivalent knowledge in the field of computer science, as that of a 10^{th} semester software-engineering student

Three kind of sources references are used throughout this report. The first is a reference placed after a period, which refers to the given paragraph. The second type of reference is placed before a period, which refers to the particular sentence or word. The last type of reference is placed after a colon or semicolon and refers to the following list. The sources of the references used throughout this report can be found in the bibliography at the end of the report.

René K. Hornbjerg
rande10@student.aau.dk

Contents

Chapter 1

Introduction

In the modern world, the number of computer systems is ever increasing, and as a result thereof, a large set of different computer systems exists.

One of the types of computer systems are *embedded systems*. Embedded systems are characterized by the platform they are designed to operate on. Embedded systems operates on a small platform with limited resources in terms of processor and memory.

Another type of computer systems are *Real-Time systems* whose reliability is determined by their ability to complete tasks before a given deadline.

Real-Time systems can be separated into three categories, depending on how they tolerate missing a deadline, *hard Real-Time*, *soft Real-Time*, and *firm Real-Time*. Hard Real-Time systems have no tolerance for missed deadlines, soft Real-Time systems tolerate missed deadlines to a certain degree, and firm Real-Time systems are a blend between hard and soft Real-Time systems. A hard Real-Time system, whose failure result in loss of human lives, is called a *Safety Critical System*.

The combination of the two types of computer systems described above is known as an *Embedded Real-Time system*. An embedded Real-Time system is required to work on a platform with limited resources, and at the same time respect timing constrains in order to function correctly.

Embedded Real-Time systems are usually implemented using low-level programming languages such as C or `Assembly`. The use of low-level languages is based on hardware controllability, as low-level languages offer more direct control over the underlying hardware compared to high-level languages such as C# and Java. Additionally, the execution time of low-level languages is typically faster than that of high-level languages.

Despite the widespread use of low-level language, these languages can be difficult to master and they are usually not taught to programmers during edu-

cation. Hence, the programming industry is faced with the problem of finding skilled programmers for embedded Real-Time systems.

1.1 The Problem

The challenge of finding skilled low-level programmers for the industry have lead researchers to focus on enabling high-level languages for use in programming embedded Real-Time systems.

One of the most popular languages in this regard is the programming language Java. Java is a high-level language following the Write Once Run Anywhere (WORA) principle. Java is taught at a large number of programming educations and is one on the popular high-level languages for most developers[8]. Unlike the low-level languages, Java is usually not natively compiled, but relies on a Java Virtual Machine (JVM) for execution. Different JVM implementations exist, however they all execute the Java bytecode generated by the Java compiler.

The JVM has been a the target for many researchers aiming at enabling Java for embedded Real-Time applications. However, general purpose implementations of the JVM, such as Oracle Hotspot[5], are not timing predictable which is a requirement if the Virtual Machine (VM) is to be used for Real-Time applications. Also these JVM implementations are usually not targeted at embedded systems.

While implementations of the JVM, targeted at embedded Real-Time systems exist, the focus of this report is the Dalvik Virtual Machine (DVM) residing inside the Android mobile Operating System (OS). Java is the primary development language for Android. Additionally, as a consequence of Android running on embedded platforms, the DVM is designed for such platforms.

The DVM is not timing predictable, as such it cannot safely be used for Real-Time applications. This raises the question of whether it is possible to make the DVM timing predictable, and if so, how it can be done.

1.2 Report Overview

The remainder of the report is structured in the following chapters:

Chapter 2: Related Work

Topics and research paper related to Java and Android with respect to Real-Time are described to cover solutions related to the problem at hand.

Chapter 3: Contribution

The essential inner workings of the Android OS and the DVM are described along with this projects approach for a solution.

Chapter 4: Conclusion

The knowledge optioned during the project is reflected upon and the project is concluded. Finally, a list of future work are presented.

Chapter 2

Related Work

This chapter presents a small list of terminology followed work related to the subjects of Java as a Real-Time programming language as well a general introduction to the DVM.

2.1 Terminology

This section presents a short list of terminology used throughout the remainder of the report.

Safe & Precise Worst Case Execution Time (WCET)

- A safe WCET is guaranteed never to be exceeded.
- The precision of a calculated WCET is determined by its deviation from the task's actual WCET.

Real-Time Garbage Collection
A Real-Time Garbage Collector (GC) is executed concurrently with the other tasks in the application. The GC can be preempted by any other task at any time if the need arises. Additionally, the GC is fully predictable in term of both behavior and execution time.

2.2 Java in Real-Time

As the Java programming language was original designed for general desktop computer systems, the system model of the language does not fit the requirements prescribed by embedded Real-Time systems. To overcome this, both the Java programming language and the JVM specification have been extended. The

most notable extensions of Java, in terms of Real-Time, are the Real-Time Specification for Java (RTSJ)[1] and Safety-Critical Java (SCJ)[3].

2.2.1 Real-Time Specification for Java (RTSJ)

The RTSJ extends the Java class library in seven areas as described in my pre-specialization[4]. The two extensions which are most important to this project are presented here.

Memory Management

The concepts of *immortal* and *scoped* memory allow for allocation of objects not affected by the GC.

Scheduling

To make scheduling comply with a program's timing predictability, the RTSJ provides the classes; *Parameter Classes* and *Scheduler Classes* along with the interface `Schedulable`. These make Real-Time scheduling within the application possible.

2.2.2 Safety-Critical Java (SCJ)

SCJ defines a more limited programming model, most notable of which is the concept of *missions*. This concept will be referenced in a discussion of timing presented later in the report. Missions consist of three phases; *Initialization*, *execution*, and *termination*. All reference types for the mission is allocated and initialized during the initialization phase. The execution phases executes the mission's tasks concurrently, and the termination phase is reached when all tasks is completed. The termination phase may include a global clean-up, after which the mission can be re-initialized.[4]

2.2.3 Hardware near Virtual Machine

The Hardware near Virtual Machine (HVM) is a software implementation of a JVM designed for bare metal use on low-end embedded devices. The HVM implements the SCJ profile and supports application of SCJ compliance level 0 and 1.[9] The HVM was at a later time re-factored in order to make the interpreter timing predictable. The re-factored HVM is referred to as Time Predictable HVM (HVM_{TP})[7].

The HVM_{TP} was constructed by analyzing the original HVM implementation of each Java bytecode with focus on timing predictability. By harnessing concepts of SCJ the bytecode implementation found to be unpredictable was re-factored, in the end leaving all bytecode implementations timing predictable.

2.3 The Dalvik Virtual Machine (DVM)

The DVM is the execution environment used in the Android OS up til Android version 4.4. The DVM targets embedded systems with limited resources such as processor speed, memory, battery, etc.[2]

The DVM differs from a classic JVM implementation in a few ways;

Execution Language

The code executed by the DVM is called *Dalvik Executable (dex) code*. The dex code is generated by a tool in the Android tool-chain, called *dexgen*. Dexgen is given Java bytecode, in form of `.class` file, as input and generates the equivalent dex code stored in a single `.dex` file.[2]

Calculations Performing

The DVM uses 32 bits registers for performing calculations. The register-based approach used for the DVM may yield a performance boost if the machines is adjusted to the underlying hardware[6]. The use of registers, in contrary to a stack, does however require longer instructions.

Applications running on the Android OS are sandboxed from each other. This means that each application is running on its own DVM instance, and each application thereby have no notion of other running applications. Each new DVM instance is provided at application start-up by the tool called *Zygote*.[6] The *Zygote* tool can be described as an "über"-DVM started at the same time as the Android OS. Each time an application is about to start, a start-up request is sent to the *Zygote* tool, and a new DVM instance is *forked* from the DVM instance of the *Zygote* tool.

Chapter 3

Contribution

This chapter describes the inner workings of the Android with focus on the execution environment, the DVM. Furthermore, this chapter presents the approach for a solution to the problem presented in Section 1.1.

3.1 Approach for Solution

This section describes the approach used for addressing the problem described in Section 1.1. The approach in this chapter is largely inspired the work of Luckow et al. presented in [7] which is also described in Section 2.2.

The problem is described as folows;

> Is it possible to make the DVM timing predictable, and if so, how it can be done.

The approach in this project is aimed at the portable version of the iterative interpreter living within the DVM. The interpreter in the DVM works in three major steps; 1) fetch, 2) decode, and 3) execute. The implementation of the main loop is illustrated in Listing 3.1.

The first step, illustrated in line 7, reads the opcode pointed to by the instruction pointer, also known as the *Program Counter (PC)*. This is done in constant time.

The second step, shown in line 9, is handled by a jump-table with a constant look-up time.

As the two first steps can both be considered timing predictable without further work, the third is left as the only obstacle. The third step is performed by performing logic starting from the jump target and this step is thus the only step which is dependent on the current context.

The logic of each jump target is referred as the *opcode implementations*. The approach here is to analyze the opcode implementations with a focus on predictable timing behavior.

Listing 3.1: The main loop for the iterative interpreter of the DVM

```
1  #define FETCH(_offset) (pc[(_offset)])
2
3  #define INST_INST(_inst) ((_inst) & 0xff)
4   // ...
5    while (1) {
6      /* fetch the next 16 bits from the instruction stream */
7      inst = FETCH(0);
8
9      switch (INST_INST(inst)) {
10       // Opcode implementations
11       // ...
12       }
```

3.2 Opcode Analysis

Opcodes operating in a similar way can be sorted into opcode groups. The opcodes in each group is implemented in similar ways or used for operating with similar concepts.

Each opcode group is sorted as *Intuitive Predictable* or *Non-Intuitive Predictable* presented in Section 3.2.1 and Section 3.2.2 respectively.

3.2.1 Intuitive Predictable Opcodes

This section presents the groups of opcode which either naturally have predictable execution time, or require simple modifications for enabling predictability.

Move Opcodes Opcodes in this group are used to move data between registers. The group include; `move`, `move-wide`, `move-object`, and `move-result`. The opcodes in this group is considered simple and includes no logic which causes problems for timing predictability.

Return Opcodes Opcodes in this group are used for returning results from methods. The groups includes; `return-void`, `return`, `return-wide`, and `return-object`. As opcodes in the `move` group, this group of opcodes does not introduce problems for timing predictability.

Goto This group of opcodes is used for unconditional jumps. The group includes; `goto`, `goto/16`, and `goto/32`. The opcodes in this group is implemented by simply adjusting the PC. Hence, the cause no problem for timing predictability.

Comparison of Primitive Type This group is used for comparison of two values of the same primitive type. The opcodes in this group include; `cmpl-float`, `cmpl-double`, and `cmp-long`. The opcodes are all handled in the same way, by the function `HANDLE_OP_CMPX(_opcode,_opname,_varType,_type,_nanVal)`. This function is simple and no unpredictable logic is used.

If Jumps This group is used for conditional jumps. The opcodes in the group include; `if-eq`, `if-ne`, `if-ge`, `if-eqz`, and `if-nez`. These opcodes are implemented in a similar way to the `goto` opcodes and no unpredictable logic is used.

Primitive Type Conversion The primitive types supported by the DVM are; `short`, `int`, `float`, `double`, `long`, `byte`, and `char`. The conversion between the types are in most cases a simple problem of altering the size of data representation, and this is considered no problem for timing predictability.

Mathematical & Bit Operations The mathematical and bit operations supported by the DVM include; multiply, division, reminder, xor, and shifts. Though these opcodes are platform dependent, they should not be a problem for timing predictability. Multiplication can, for example, be conducted by using addition and shifts with an execution time bounded by the number of bits.

Switch This group of opcodes handled switch statements. The opcodes in this group are; `packed-switch` and `sparse-switch`. Both of these opcodes are implemented in a simple way using a jump table which is considered to have constant time look-up time. Thus, this opcode group is of no concern for timing analysis.

3.2.2 Non-Intuitive Predictable Opcodes

This section presents the groups of opcodes which require re-factoring in order to enable timing predictability. Groups which call for further reflection are discussed in Section 3.3.

Constant Opcodes Opcodes included in this group are; `const`, `const-wide`, `const-string`, and `const-class`. The opcodes for handling constants of primitive types are of no concern in timing perspective. However, the implementation of constant strings and classes could cause problems. The implementations of `const-string` and `const-class` are almost identical.

The implementations of the opcodes poses no obvious problem. However, the use of constants of non-primitive types raises issues as the object for the constant reference is to be initialized before the constant can be used. The time of instantiating may be unknown at start-up time. A work around for this could be to always assume that all objects needs to be instantiated before the can be used. This would yield a safe WCET, the WCET would, however, not be precise and it would in most cases be a heavy overestimation. This issues is discussed further in Section 3.3.

Type Checking Opcodes This group is used for run-time type checking of reference types. The group includes the two opcodes; `instance-of` and `check-cast`. In classic JVM implementations these operations is handled by iterating through the class hierarchy until compatibility can be concluded. In the DVM, these operations are handled by the `dvmInstanceof()` function. The implementation of `dvmInstanceOf()` is shown in Listing 3.2.

The implementation makes used of other defined functions, such as `dvmInstanceOfNonTrivial()`. The actual workings behind the type checking is unclear for the author, so whether type checks is handled by iterating through the class hierarchy is also unclear. It is, however, unlikely this way it is done. As the complete class hierarchy can be found in the dex file at compile-time, it is most likely that the DVM is implemented to use this for optimization purposes.

Object Allocations The group includes; `new-instance`, `new-array`, `filled-new-array` and `filled-new-array-range`. When an object is allocated in the DVM, the memory for the object is zeroed to avoid bad memory values. The execution time for zeroing the memory is linear in the size of the object. A WCET corresponding to the largest object in the application can be found. However, thought this WCET can be considered safe for all allocations, it is in many cases not precise and it is therefore desirable to find an other approach. Further discussion on this is given in Section 3.3.

Gets & Puts This group of opcodes is used for getting and setting values of different types into arrays, instance fields, and static fields. This group of opcodes is implemented using several general defined functions. As for this constant opcodes, the implementation of these opcodes poses no obvious problem. How-

ever, as for the constants of reference types, getting an setting data to a static field requires the static object to be initialized. Assuming initializing is needed every time a static field is called would yield a safe, but rather inaccurate, WCET. This problem does not occur for instance field as each instance is assumed to be initialized before any values can set of received from the instance. This issues is discussed further in Section 3.3.

Method Invocation This group of opcodes includes; `invoke-virtual`, `invoke-super`, `invoke-static`, and `invoke-interface`. Management of method invocation is handled by the use of a call stack which introduces no obvious issues for timing predictability.

Implementation of all the previous opcodes a done in similar ways. All opcodes uses a function called `invokeMethod()`. This function handles coping arguments from the current stack frame to the new stack frame, updates the required values (PC, Frame Pointer (FP), etc.) and jumps to the start of the interpreter. Hence, this function is responsible for transferring control which causes no problem for timing.

Methods is stored in a *virtual table*. Each method can be found by indexing the table with the methods index, which is a constant time operation. `invoke-virtual` and `invoke-super` implements this in similar ways. `invoke-interface` uses a slightly different approach, the function `dvmInterpFindInterfaceMethod()`. This function iterates the *if-table* of the receiver of the method call, which is the only operation in this function without constant execution time. The execution time for the operation of iterating through the if-table is linear in the size of the if-table. The size of the if table is known at compile-time, which makes the overall execution time of the function bound-able at compile-time.

Exceptions Exceptions can be thrown in one of two forms, unchecked or checked. Unchecked exceptions are thrown implicit by the opcode implementations. Checked exceptions are thrown explicit by use of the `throw` opcode.

One issue raised by use of exceptions are similar to the issues of constant reference types and manipulation of static fields. Although a safe WCET corresponding to the initialization of the largest exception object can be found, this is still an inaccurate and heavy overestimated WCET. Further discussion on this is given in Section 3.3.

The second issue raised by use of exception is determining and execute the handler for the exception. The exception may be caught somewhere in the call stack. Assuming the worst-case behavior, the maximum size of the call stack is used. The maximum size of the call stack can be determined by a reconstruction of the call graph, and then determined the maximum depth. Reconstruction of

the call graph and maximum depth determination would require the use of a tool. In the case of the HVM_{TP}[7] the ICECAP-TOOLS is used for this matter.

Remaining Opcodes Some of the opcodes implemented in the DVM are considered only for advanced use or only for use on optimized dex files. Those opcodes are not included in this analysis as they are considered dispensable. The author of this report is of the believe that a fully functional DVM could be implemented without the use of those opcodes. This DVM version would though require further verification of the dex file to make sure the dispensable opcodes is not used.

The art of making these opcode timing predictable should, however, be a simple manner of using the same mindset as for the other groups of opcodes.

Listing 3.2: Implementation of `dvmInstanceOf()`

```
1 INLINE int dvmInstanceof(const ClassObject* instance, const ClassObject*
      clazz)
2   {
3     if (instance == clazz) {
4       if (CALC_CACHE_STATS)
5       gDvm.instanceofCache->trivial++;
6       return 1;
7     } else
8     return dvmInstanceofNonTrivial(instance, clazz);
9   }
```

3.3 Timing Discussion

This section presents a discussion of the issues with timing predictability found during the opcode analysis described in Section 3.2.

An issue which was encountered several times during the analysis resolve around the time of which an object is initialized. Due to the standard Java programming model used by the DVM, allocation of a new object can be tricked, roughly, at any time during execution of an application. If a WCET for the functions in an application are to be calculated, the worst-case behavior must be assumed at all times, and in the case of object allocations, the worst-case behavior would be to initialize a new object every time an object is referenced.

Analyzing the program flow before execution could mark every occurrence of referencing and thereby incorporate the WCET for object initialization into the WCET of each function. Thought this approach would yield a safe WCET for all functions, the WCETs would in many cases be highly inaccurate.

Introducing a different programming model similar to that of SCJ would help decrease the inaccuracy. Using a programming model dictating all objects to be allocated prior to the logic of the application would decrease the inaccuracy greatly. Allocation all objects prior to application logic would yield a different worst-case behavior for functions, as object in this context can be assumed to be allocated at every reference.

The accuracy problem would, however, only be partly solved. The WCET for object allocation would still be corresponding to the largest object of the applications, which is not accurate for all objects. This issue is resumed later in this section.

Deallocation of object is another issue. As the behavior of the DVM's default GC is highly unpredictable, deallocation can happen at practically any time. Ad-

ditionally, the GC stops all execution of the application to keep the memory in a consistent state.

A simple way to overcome this issue would be to disable the GC. This would, however, introduce a risk of running out of memory. An other way to overcome this would be to introduce a predictable GC as described in Section **??**.

The issue of inaccurate WCETs for object allocations is due to the zeroing of memory at allocation time. The execution time for zeroing the memory is linear to the size of the object, yielding a allocation WCET corresponding to the largest object in the application.

Relocating the responsibility for zeroing of memory would partly solve the issue. Zeroing the memory at system start-up, that is before the application is started, would increase the system start-up time, but would remove the issue at application run-time.

The remaining part of the issue revolves around deallocation. If the approach of disabling the GC is used, memory would not require a re-zeroing as it is never freed. Using a predictable GC memory could be re-zeroed subsequently to freeing, thereby extending the execution time of deallocation, but it not influencing allocation of new objects.

Re-factoring the DVM in the correct way may make it timing predictable, but even a timing predictable version of the DVM would not be completely suitable for Real-Time systems. The DVM is designed to run on top a Linux kernel. The Linux kernel used for the Android OS, and thereby the DVM, is not predictable in a way which support Real-Time applications.

Two ways for addressing this issue is considered; replacing the default kernel with a Real-Time kernel, and removing the need for the kernel

Substituting the standard kernel with a Real-Time version would make the DVM better suited for use for Real-Time applications. A complete replacement of the kernel may, however, require modifications of the interface between the DVM and the kernel. Additional work might be needed for porting the drivers from the default kernel to the new Real-Time version. However, if the DVM is used as a stand-alone VM for Real-Time many of the drivers required for the Android OS would not be needed. In this case, the embedded platform can be assumed to be more specialized for a specific application domain thereby not requiring the presence of, for example, Wifi, touch-screen, etc.

Removing the kernel would lead to a bare-bone version of the DVM. This would be beneficial in terms of system size. If the DVM does not require the

presence of a OS the overall system size would be greatly reduced. The approach would require a extension of the VM. Lacking the presence of an OS the responsibility for systems facilities such as Input/Output (I/O) and locking of memory would have to be handled directly in the VM. The same is the case for scheduling.

Discussion Summary

The problem of WCET for object allocation yielding a rather inaccurate WCET for the overall system, was encountered several times during the analysis of the opcodes. Seeking inspiration from SCJ, introduction of a programming model dictating allocation of all object prior to logic execution would reduce, but not remove, the problem. Relocating the responsibility of memory zeroing in combination with a disabled GC would help eliminating the problem, but disabling the GC would introduce the risk of running out of memory.

Adding a different programming model and making all opcodes timing predictable would reduce the unpredictability of application execution on the DVM, but the DVM would still not be suitable for Real-Time systems. Enabling the DVM for Real-Time usage would require attention to the underlying Linux kernel. Replacing the kernel with a Real-Time version would enable the DVM for Real-Time applications, but adaption between the DVM and the new kernel would properly be needed. Removing the kernel from the picture could also enable the DVM for Real-Time usage, but would require an extension of the DVM as it would need to overtake the responsibility of system facilities such as I/O, locking of memory, and scheduling.

Chapter 4

Conclusion

This project have introduced the problem of lack of developers for embedded Real-Time systems. Attempting to solve this problem, research in the usage of high-level languages for Real-Time programming have been conducted. This spans from the introduction of new programming models for Java such as the Real-Time Specification for Java and the Safety-Critical Java profile, to introduction of Java Real-Time platforms such as the Time Predictable HVM.

This project have focused on introducing the Dalvik Virtual Machine as a competitor to the already existing Java Real-Time platforms. Although, the DVM is not implemented in the same way as classic Java Virtual Machines, the DVM still relay on Java a primary programming language.

This report have presented an analysis and discussion of DVM with focus on timing predictability. The analysis targeted the opcode implementations of the portable iterative interpreter in the DVM. Analyzing the individual implementations of the opcodes yielded two major groups of opcodes; *Intuitive Predictable* and *Non-Intuitive Predictable*. The analysis found that some of the opcode implementations resulted in work additional to a re-factoring of the implementations.

The discussion revolved around the additional required work found in the analysis. Encountering a few problems in relation to object allocations called the need for a different programming model to mind. Introducing a programming model dictating allocation of object prior to execution of logic would increase the probability of introducing the DVM as a new Java Real-Time platform.

However, introducing a new programming model alone, would not make the DVM suitable for Real-Time usage. The underlying Linux kernel would require attention as well.

Based on the analysis and discussion summarized above, it is concluded that it would be possible to construct a timing predictable version of the Dalvik Virtual Machine. However, attention only to the opcode implementations would not be sufficient.

4.1 Future Work

This section presents the future work for this project. The topics presented in this section mainly focus on what would be required to construct a Real-Time version of the DVM.

Stand-alone DVM

The first step towards a Real-Time version of the DVM would be to extract the DVM and its core dependencies from the Android system. Once extraction have been done successfully drivers are no longer of concern for the kernel.

Replacing the Linux Kernel

The default kernel shipped along side the Android OS is not suitable for Real-Time requests. Replacing the kernel with a Real-Time version with the required system facilities placed along side a stand-alone DVM would simplify the remaining work needed for a Real-Time DVM.

Removing the Linux Kernel

With a stand-alone version of the DVM at hand, construction of a bare-bone version of the DVM is right up our street. Constructing a bare-bone version would reduce the overall size of the execution environment. On the other hand, the DVM version would need to be extended with system facilities such as scheduling and I/O management.

List of Listings

Bibliography

[1] Greg Bollella, Ben Brosol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*, 2013. URL https://jcp.org/aboutJava/communityprocess/first/jsr001/rtj.pdf.

[2] David Ehringer. The davlik virtual machine architecture, March, 2010. URL http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

[3] The Open Group. *Safety-Critical Java Technology Specification*, 2013. URL http://download.oracle.com/otn-pub/jcp/safety_critical-0_94-edr2-spec/scj-EDR2.pdf.

[4] René Kjær Hornbjerg. Android in real-time. Student report, 9. semester, Aalborg University, 2014.

[5] Hotspot. Hotspot. URL http://en.wikipedia.org/wiki/HotSpot.

[6] Igor Kalkov, Dominik Franke, John F Schommer, and Stefan Kowalewski. A real-time extension to the android platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 105–114. ACM, 2012.

[7] Kasper Søe Luckow, Bent Thomsen, and Stephan Erbs Korsholm. Hvmtp: A time predictable and portable java virtual machine for hard real-time embedded systems. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, 2014.

[8] Tiobe Software. Tiobe index for january 2015, 2015. URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[9] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. Safety-critical java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, 2012.

Resume

Dette projekt påpeger et problem med mangel på udviklere til *Indlejret Tidstro Software*. I forsøget på at løse dette problem er der blevet forsket i brugen af høj-niveau sprog for tidstro programmering. Forskningen strækker sig fra, indførelsen af nye programmerings modeller for Java såsom RTSJ og SCJ profilen, til indførelsen af tidstro Java platforme såsom HVM_{TP}.

Fokuset is dette projekt har været på at indføre DVM'en som konkurrent til de allerede eksisterende tidstro Java platforme. Selvom DVM'en ikke er implementeret på samme måde som de klassiske JVM'en, bruger den Java som det primære programmeringssprog.

Denne rapport præsenterer en analyse og diskussion af DVM'en med fokus på tidsforudsigelighed. Analysen er målrettet implementeringerne af opkoderne i den iterative fortolker. Analysen af de individuelle opkode implementeringer giver to hovedgrupper; *Intuitivt Forudsigelig* og *Ikke Intuitivt Forudsigelig*. Analysen fandt, at nogle af implementeringerne behøver arbejde udover en refakturering.

Diskussionen tager udgangspunkt i det supplerende arbejde fundet i analysen. Op til flere gange i analysen fremkom et problem i forhold til allokering af objekter. Dette problem ledte tankerne hen på behovet for en anderledes programmeringsmodel. Indførelse af en programmeringsmodel som dikterer at alle objekter skal allokeres før applikationens logik kan eksekvere ville øge muligheden for at indføre DVM'en som en ny tidstro Java platform.

Derudover, dækker diskussionen nødvendigheden for at kigge på den underliggende Linux kerne som ikke er brugbar til tidstro software.