Evaluation of a hardware accelerated Java Virtual Machine on embedded devices

Anders Holm
Anders Mårtensson

November 24, 2006
Abstract

Java is a technology which is used in billions of products, such as desktop computers, servers, and mobile phones. It has many popular features; it is platform independent, it has automatic memory management (garbage collection), it is object oriented, it has a good security model, and it has a good API.

Java has some performance problems though. Execution speed isn’t as good as it could be since bytecodes are interpreted and translated in software and then executed in hardware. Execution smoothness is far from optimal – desired Java techniques such as garbage collection and just-in-time compilation have a negative influence on execution smoothness. If a just-in-time compiler is used, startup time may be increased. Memory consumption is also very important on embedded devices with limited resources, such as a mobile phone.

These problem areas are investigated to see if they can be reduced by using a hardware accelerated Java virtual machine in an embedded system (a mobile phone). A model for measuring execution smoothness is proposed and discussed. It is also verified and used to measure execution smoothness in games.

Tests show that a hardware accelerated Java virtual machine is almost as fast as a just-in-time compiler enabled Java virtual machine. Startup time is decreased compared to a just-in-time compiler enabled Java virtual machine. Execution smoothness is improved, which is especially important in games, and memory usage is also decreased.

The model for measuring execution smoothness was proved to work well and makes it possible to compare graphs not only visually, but also numerically.

Keywords: Java hardware acceleration, Java performance issues, execution speed, execution smoothness, startup time, benchmarking, Java on embedded devices.
Acknowledgements

Thanks to (in alphabetical order) Anders Nilsson (our supervisor at Ericsson AB, for guidance), David Petterson (Ericsson AB, for help fixing coding/compilation/linking errors), Sven Gestegård-Robertz (our supervisor at LTH, for input on the report) and Torsten Bögershausen (Ericsson AB, for help during debugging) and to the rest of the Java team at Ericsson AB.
Contents

1 Introduction .................................................. 1
   1.1 Project description ....................................... 2
   1.2 Scope .................................................. 2
   1.3 Intended readers ......................................... 2
   1.4 Report outline .......................................... 2

2 Theory ........................................................... 5
   2.1 The Java 2 Platform Micro Edition .................... 5
   2.2 The Java Virtual Machine ................................ 6
      2.2.1 The architecture of the Java virtual machine .... 7
      2.2.2 Implementations of the Java virtual machine .... 11
   2.3 Java hardware acceleration technologies ................ 12
      2.3.1 Dedicated Java processors .......................... 12
      2.3.2 Java co-processors .................................. 12
      2.3.3 Architectural extensions ............................ 12
   2.4 Areas of interest .......................................... 13
      2.4.1 Execution speed ..................................... 13
      2.4.2 Execution smoothness ............................... 14
      2.4.3 Memory usage ....................................... 14
      2.4.4 Startup time ........................................ 15

3 Methods to measure execution smoothness ..................... 17
   3.1 Definition of smoothness ............................... 17
3.2 Measuring smoothness ........................................ 19
  3.2.1 Existing models ........................................ 19
  3.2.2 Our model ........................................ 19
3.3 Verifying the model – a survey .............................. 23

4 Testing .................................................................... 25
  4.1 Test configurations ........................................ 25
  4.2 Testing methods ............................................... 26
    4.2.1 Benchmarking execution speed .................... 26
    4.2.2 Benchmarking startup time ....................... 26
    4.2.3 Benchmarking execution smoothness ........... 27
    4.2.4 Benchmarking memory usage .................... 27
  4.3 Test applications ........................................... 27
    4.3.1 Amark 1.3 ........................................... 27
    4.3.2 JBenchmark 2 ....................................... 28
    4.3.3 EEMBC GrinderBench ................................ 28
    4.3.4 Racing Fever 2 ....................................... 29
    4.3.5 Extreme Air Snowboarding ....................... 29
    4.3.6 Racing Fever 3D ....................................... 29
    4.3.7 3D Extreme Air Snowboarding .................... 30

5 Test results and discussion ................................... 33
  5.1 Execution speed ........................................... 33
  5.2 Startup time ............................................... 35
  5.3 Execution smoothness ..................................... 36
  5.4 Memory usage ............................................. 38

6 Conclusions and further work ................................ 39
  6.1 Conclusions ................................................ 39
  6.2 Further work ............................................... 41
    6.2.1 Other hardware accelerators .................... 41
    6.2.2 Other Java virtual machines ..................... 41
6.2.3 Other domains .................................................. 41
6.2.4 Power consumption ........................................... 42
# List of Figures

2.1 The J2ME building blocks .................................. 6
2.2 Architecture of a Java virtual machine ................. 7
2.3 Class loading, linking and initialization ................. 8

3.1 A rough graph both locally and globally ................. 17
3.2 A smooth graph both locally and globally ................. 18
3.3 A graph which is rough globally, but is smooth locally 18
3.4 Description of our model .................................. 20
3.5 Our smoothness model on data set 1 and 2 (synthetic) 21
3.6 Screenshot from the test MIDlet used to verify the smoothness model 23

4.1 Screenshots from Racing Fever 2 .......................... 29
4.2 Screenshots from Extreme Air Snowboarding ............. 30
4.3 Screenshots from Racing Fever 3D ........................ 30
4.4 Screenshots from 3D Extreme Air Snowboarding ........ 31

5.1 Execution speed test results (JBenchmark 2) ............. 33
5.2 Execution speed test results (Extreme Air Snowboarding) 34
5.3 Execution speed test results (EEMBC GrinderBench) .... 34
5.4 Startup time test results ................................... 35
5.5 Execution smoothness test results (Extreme Air Snowboarding) 36
5.6 Execution smoothness comparison between different VM configurations 37
5.7 Memory usage test results (3D Extreme Air Snowboarding) 38
List of Tables

3.1 Results from our model on two data sets. ................................. 22
3.2 Survey results to verify that the model for measuring smoothness ... 24
## List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOT</td>
<td>Ahead Of Time compiler</td>
<td>9</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
<td>6</td>
</tr>
<tr>
<td>CDC</td>
<td>Connected Device Configuration</td>
<td>5</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
<td>5</td>
</tr>
<tr>
<td>DAC</td>
<td>Dynamic Adaptive Compiler</td>
<td>9</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
<td>10</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java 2 Platform, Micro Edition</td>
<td>5</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time compiler</td>
<td>9</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
<td>28</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td>6</td>
</tr>
<tr>
<td>MIDlet</td>
<td>Mobile Information Device Applet</td>
<td>25</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
<td>6</td>
</tr>
<tr>
<td>WAT</td>
<td>Way Ahead of Time compiler</td>
<td>9</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Java is a technology which is spread and used more and more for each day [9]. It has found its way to new devices, for instance mobile phones. As of June 2005, 708 million Java-enabled mobile devices have been sold [13]. The reasons why many developers like Java are many. To mention a few, the write-once run-anywhere model, portability, a clean language, a good API, and a good security model are all examples of good and desired features of Java technology.

Unfortunately, Java comes with some performance problems. The problems that this report will focus on are; execution speed, execution smoothness, startup time, and memory usage.

Speed is an important aspect of applications. Users expect applications to run fast, which means calculations being performed as fast as possible and as many displayed frames per second as possible in games. Users also expect games to run smoothly without annoying slowdowns every now and then. Startup time must be very short because most people expect an application to start very fast, for instance in less than one second. Memory usage is particularly important in embedded environments with memory constraints.

The idea of using hardware acceleration is a common technique to make slow things faster. It can be found in different areas such as accelerating math calculations using a Floating Point Unit (FPU), accelerating real-time 3D graphics using a Graphic Processing Unit (GPU) or, as in this thesis, accelerating Java through the use of an added instruction set.

The use of Java hardware acceleration may be a way to reduce or remove some of the problems described above.
1.1 Project description

The aim of this project is to evaluate how the integration of a Java hardware accelerator into a Java virtual machine will affect:

- Execution speed
- Execution smoothness
- Startup time
- Memory usage

Since no de-facto model for measuring execution smoothness exists, a new model for this is constructed. The model is then verified and used to measure execution smoothness in games.

The thesis was conducted at Ericsson AB, Business Unit Mobile Platforms, in Lund, Sweden.

1.2 Scope

Only one hardware acceleration solution is evaluated. This report isn’t meant to be a comprehensive guide to all existing hardware solutions, but instead to show what hardware acceleration may improve in general.

Only one Java virtual machine is used. There exists several other Java virtual machines, and the results presented in this report may or may not be valid with other Java virtual machines.

1.3 Intended readers

This report is aimed at wide readership of students and IT professionals. An ideal reader is a software developer or project leader/manager in the domain of embedded systems and Java virtual machines. The report is written so that it can be fully understood by students and professionals who posses more than basic knowledge of Java, but no expert knowledge is required. For most readers, this corresponds to the third or fourth year of a university degree in computer science.

1.4 Report outline

This report is divided into five parts:
1.4. Report outline

- **Theory.** Chapter 2 gives the necessary theoretical background. Section 2.2 describes the basics of how Java Virtual Machines work. The areas of interest, are described in section 2.4.

- **Methods to measure execution smoothness.** Chapter 3 discusses the theory of measuring smoothness and a new model is presented.

- **Testing.** Chapter 4 describes how the tests were performed and presents the MIDlets (i.e. Java applications for J2ME) that were used.

- **Test results and discussion.** Chapter 5 presents the results of the tests. Focus are on the areas execution speed, execution smoothness, memory usage, and startup time. The results are analyzed and discussed.

- **Conclusions and further work.** Chapter 6 gives a brief summary of the work and the results. Other interesting aspects of hardware acceleration that was out of the scope for this thesis are also discussed.
Chapter 1. Introduction
Chapter 2

Theory

This chapter is intended to give an introduction to the technologies discussed and used during this thesis.

It’s necessary to understand the inner workings of a Java virtual machine to be able to understand the reasons behind the problems discussed in this thesis. The areas of interest are stated and the reasons why they are interesting are motivated.

2.1 The Java 2 Platform Micro Edition

The Java 2 Platform, Micro Edition (J2ME) [22] is a platform that is adapted to devices such as mobile phones, PDAs and other customer devices or embedded devices, all of which have limited resources. Other editions of the Java platform are the Enterprise Edition (J2EE), Standard Edition (J2SE) and Java Card. J2ME includes Java virtual machines and a set of standard Java APIs which are defined through the Java Community Process (JCP) [31]. JCP consists of leading device manufacturers, software vendors and service providers.

The J2ME architecture consists of a variety of configurations, profiles, and optional packages that can be chosen depending on the target device, see figure 2.1. Each combination is adapted to available memory, CPU resources, and I/O capabilities.

Configurations consist of a virtual machine and a minimal set of class libraries. They provide the base functionality for a particular range of devices that share similar characteristics. There are two J2ME configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC).

To complete the runtime environment for a specific device category, a configuration must be combined with a profile. A profile is a layer on top of the configuration that provides additional APIs for a specific class of devices. Profiles support a narrower category of devices within the framework of a chosen configuration. An example is
to combine CLDC with the Mobile Information Device Profile (MIDP) to provide a complete Java application environment for mobile phones and other devices with similar capabilities.

### 2.2 The Java Virtual Machine

This section will henceforth focus on the JVM and what goes on inside. To read more about the Java programming language see the Java Language Specification [14].

The JVM is the heart of the Java technology – it’s an abstract computer on which all Java programs run. The Java Virtual Machine Specification [15] defines features that every JVM must support. These definitions aren’t implementation specific but rather describes what the JVM must support and not how this support should be implemented.

When a Java application is started on a device, a runtime instance of the JVM is created. It has a single job – to run the application. When the application terminates, the instance of the JVM that was running the application also terminates (depending on the implementation).

When a Java application is started the initial classes and interfaces used in the program are located and loaded – into the JVM. During the execution of the program the JVM will locate and load classes and interfaces that are needed, this process is called dynamic linking. All classes keeps symbolic references in the constant pool,
2.2. The Java Virtual Machine

when a reference is needed it must be resolved. Resolution is the process of replacing a symbolic reference with a direct reference.

2.2.1 The architecture of the Java virtual machine

Figure 2.2: Architecture of a Java virtual machine. The components displayed in gray are explained and investigated in this report.

Figure 2.2 shows the internal architecture of a JVM. These subsystems and memory areas are described in the Java Virtual Machine Specification. The purpose of these components and subsystems is to define the required behavior of a JVM. The implementation of a JVM doesn’t have to be designed in this way, although it must follow the behavior described.

The class loader subsystem loads data types and classes from class files into the JVM. The execution engine is responsible for executing instructions for the methods in the loaded classes.

The class loader subsystem

When a class is loaded, a lot of things are going on behind the scenes. The class loader subsystem is responsible for finding and loading types from class files, verifying the loaded classes, allocating and initializing memory for class data, and assist in resolution.
Figure 2.3: Class loading, linking and initialization

- **Loading** — The binary data from the class file is parsed and an internal data structure is created in the method area. An instance of `java.lang.Class` is created that represents the class.

- **Linking** — Linking consists of three parts:
  - **Verification** — When a class has been loaded it must be verified to ensure that it’s correct according to the Java language and JVM. A class must be verified before it’s initialized.
  - **Preparation** — During preparation, memory is allocated for the class variables and they are set to default values (value depends on the type). Memory for data structures may also be allocated.
  - **Resolution (optional)** — During resolution classes, interfaces, fields, and methods that are referenced in the constant pool of the type are located, and symbolic references are replaced with direct references.

- **Initialization** — Initialization is the process that sets the initial values, chosen by the programmer, to the class variables. These values are collected by the Java compiler and placed in a special method that is called by the JVM.

Since all of this is performed when a class file is loaded, startup time can be long.

**The heap**

The heap is shared among all JVM threads. The heap is the runtime data area from which memory for all object instances is allocated. The size of the heap can be static or dynamic. The JVM has an instruction to allocate memory from the heap but no instruction to free memory. To reclaim memory that is no longer referenced by the running program, a garbage collector is used. The garbage collector reclaims the memory used by unreferenced objects, and in some implementations move objects to decrease heap fragmentation. No specific garbage collector algorithm is specified in the Java Virtual Machine Specification and the choice is up to the designers of the JVM.
Execution engine

The execution engine is the core of the JVM. A bytecode stream is a stream of instructions to the JVM. Each instruction consists of a bytecode and for some bytecodes also operands. All the bytecodes can be represented in 8 bits. The instructions may use entries in the operand stack, constant pool or local variables to execute an instruction. Java has chosen a stack-centered approach instead of register-based, used in many processors. This makes it easier to implement a JVM on different architectures. The Java Virtual Machine Specification simply states what should be done when a certain instruction is encountered. How this is done is up to the designer. There exists many execution techniques for executing bytecodes.

The JVM has an interpreter loop that maps bytecodes one by one into native instructions which are executed. The implementation is rather straightforward, easy to implement and easy to port to a new platform. The main drawback is that it’s slow [10]. Although this approach delivers poor performance it can be suitable in environments with little memory due to its small footprint.

Instead of interpreting bytecodes one by one, the *Just-In-Time compilers* (JIT) compile a block of bytecodes into native instructions when the block is called. The compiled native code is thereafter used when the same block is called later in the execution. JIT compilers typically offer a better performance than interpreters [10], however there are some problems. The biggest problem with JIT compilers is the large amount of memory required to store the native code that has been compiled. The compiled code is stored on the heap. Another problem is that compilation and optional optimization steals processor time from the application. This overhead can be significant when executing small or short-lived Java applications.

Most programs spend 80 to 90 percent of their time executing 10 to 20 percent of the code [11]. This fact is used by *Dynamic Adaptive Compilers* (DAC). Bytecodes are first interpreted, a profiler records the run times for each method and when a time consuming method is found, this method is compiled and optionally optimized. The next time this method is called, the native code is used instead. The profiler continuous to monitor throughout the lifetime of the program and methods that were compiled can be removed if they are found to be less time consuming at a later stage of the execution. A DAC can be said to be a JIT compiler with added intelligence, hence it’s hard to draw a clear line between them. A DAC is sometimes called JIT with adaptive optimization. DAC generally offers better performance and smaller footprint than a JIT compiler [11].

When using *Way Ahead of Time compilers* (WAT), all code is compiled and verified prior to execution. The code can be compiled and verified when developed or when loading it to the target device. The compiled code is then stored on the target device for execution. This means that startup time is fast, but the pre-compiled applications need much more storage space.

When using *Ahead Of Time compilers* (AOT), all code is compiled on target device before execution. Time of compilation is implementation specific. It can for instance be compiled at installation. It can also be compiled when the application is executed.
for the first time. This means that startup is very slow the first time, but sequential invocations have short startup times. A lot of storage space for the pre-compiled code is needed.

**Garbage collection**

The Java Virtual Machine Specification defines that heap space for objects is reclaimed by an automatic storage management system, called a *garbage collector* (GC). How this is done is up to the designer.

Garbage collectors remove the task from programmers of explicitly freeing objects (for instance with `free/delete` in C/C++). The process of explicitly freeing allocated memory can be tricky. Incorrect freeing of memory leads to programs that leaks memory. The use of garbage collectors can lead to increased productivity for programmers and more robust programs [11]. There aren’t only advantages by using a garbage collector. One problem is the overhead added which can affect program execution. Another is that it’s hard, even impossible, to know when the garbage collector will kick in and how much CPU time it will use.

Every implementation of a garbage collector must be able to detect garbage objects and reclaim the heap space used by them. Garbage objects are objects without references. Most implementations also try to decrease heap fragmentation.

*Reference counting* is a technique where a reference count is maintained for each object on the heap. When a reference is added to an object the reference count is increased and when a reference is deleted it’s decreased. When the reference count of an object is zero it’s garbage collected. This technique can run in small chunks of time and can therefore be suitable to use in real time environment where garbage collector interrupts must be small.

*Tracing collectors* trace the graph of object references to find unreferenced object. The trace is started from the root nodes. The objects that are reachable from the root nodes are called live objects. These objects are marked when tracing through the graph. When the trace is completed, the unmarked objects are garbage collected. The basic algorithm is called “mark and sweep”. Mark and sweep collectors often use a compacting technique or a copying technique to reduce heap fragmentation.

*Compacting collectors* slide objects toward one end of the heap. This was the original technique used, now there exists several better techniques. The other end of the heap becomes a free continuous space. All references to the objects are updated, sometimes this is made simpler by adding a table of object handles that refers to the object on the heap. The object references in the application refers to the object handle in the table instead of directly to the heap. When a reference is updated, only the object handler in the table must be updated. This removes the overhead of updating all references but adds overhead to every object access.

*Copying collectors* move all live object to a new area where they are placed side by side. The old area is now a free space which is used during the next collection. This approach traditionally divides the heap into two regions. The copying can be done
while traversing the nodes and thus eliminating the the sweep phase. Since only half of the heap space can be used, the memory needed for this approach is twice the memory for a given amount heap space.

*Generational collectors* take advantage of the fact that most object created are short lived and only a few are long lived. The objects are grouped by age and the younger objects are collected more often than the old. When a young object have survived a few collections it’s moved to an older generation. The heap is divided into regions that represent the age of the objects. This can be used by both mark and sweep and copying algorithms.

*Adaptive collectors* take advantage of the fact that some algorithms works better in some situations and other algorithms in other situations. The adaptive algorithm monitors the situation and chooses the algorithm that suites the situation best.

*Real-time collectors* suitable for real-time systems (or other domains where smooth execution is desired) use techniques such as *incremental collection*. Unlike many GC algorithms that must either fully complete their pass through the memory, or start over from the beginning if they are interrupted, an incremental GC works in short steps and thus allows for interruptions in between the steps.

### 2.2.2 Implementations of the Java virtual machine

In this section some examples of commercial implementations of the JVM are described briefly. Note that this isn’t a complete list, but only covers the most common JVMs for limited devices such as mobile phones.

**KVM**

The *K virtual machine* (KVM) [22] from Sun Microsystems implements the requirements of the CLDC specification [29]. It’s a JVM with a small footprint and the "K" in KVM stands for "kilobytes", referring to rather kilobytes instead of megabytes. KVM is meant to serve as a reference design that demonstrates what one implementation may look like.

**CLDC-HI**

The *CLDC HotSpot Implementation* (CLDC-HI) [20] is an optimized JVM from Sun Microsystems which implements the HotSpot technology [21]. It offers better performance than KVM and is optimized regarding performance and footprint.

It uses a dynamic adaptive compiler (described in *Execution engine*, section 2.2.1) to compile the most frequently used, time-critical parts of applications into native code. These parts are called "hot spots" and the compiler finds them by running a statistical profiler. This is a very short description of the HotSpot technology (see the Java HotSpot Virtual Machine [21] for details).
Jbed CLDC

The *Jbed CLDC* [23] is delivered by Esoteric. It’s an optimized JVM for J2ME devices. *FastBCC* is a feature in Jbed that compiles all Java code into native code. It enables fast execution speed and fast start-up time. *FastDAC* is used to deliver better Java performance than an interpreter-only JVM (KVM). All Java bytecodes initially run through the interpreter. Once a code segment has executed more than a certain number of times, it’s flagged as a candidate for compilation.

Other JVMs

There exists other JVMs as well. Two examples are *JBlend Micro* [24] from Aplix and *J9* [25] from IBM.

2.3 Java hardware acceleration technologies

There exists several different technologies and strategies for accelerating Java in hardware. This isn’t meant to be a complete list over all chip manufactures, but is instead meant as an introduction to the different technologies.

2.3.1 Dedicated Java processors

A dedicated Java processor is a processor that only runs Java bytecodes directly and doesn’t need support from a host processor. Performance is good since it has its own cache. While this may seem like a good solution, the downside is that it needs an additional processor to run existing operating systems and applications. An example of a dedicated Java processor is *aJ-100* [16] produced by Zafire.

2.3.2 Java co-processors

A Java co-processor is a processor that works side by side with an existing processor. It can for instance translate Java bytecodes into assembler instructions to be executed by the other processor. One downside is that it is usually difficult to integrate into existing operating systems [1]. It also requires additional space and power. Another problem is that it’s not implemented inside the cache, which means lower performance. An example of a Java co-processor is *JXtreme* [17] by Synopsys (former inSilicon).

2.3.3 Architectural extensions

By extending the existing *Instruction Set Architecture* (ISA) to support Java bytecodes, it’s possible to use only one processor for executing both Java bytecodes and
2.4 Areas of interest

There are four different areas which this report will focus on. The areas are: execution speed, execution smoothness, memory usage and startup time. Each of these areas will be described in this section. For each area, the major technologies causing performance problems will be discussed\(^1\). Some of the areas will counteract each other and depending on the constraints in the environment, different approaches might be preferred (for instance, more memory might be required to improve execution speed).

2.4.1 Execution speed

Execution speed is an important aspect of applications. Users expect applications to run fast which means calculations being performed as fast as possible and as many displayed frames per second as possible in games. There are many things that can affect the execution speed of an application.

Compilation

Which technology the JVM’s execution engine uses is an important factor. Using a compiler based JVM instead of a pure interpreting JVM will lead to higher execution speed. Compiler based JVMs also need to interpret bytecodes at least once, and this can be slow. If for instance a DAC is used, bytecodes usually have to be interpreted a couple of times before a code segment is considered for compilation. Hardware acceleration might lead to a higher execution speed by executing the bytecodes in hardware instead of interpreting them, perhaps without increasing the memory footprint.

Garbage collection

The choice of garbage collector algorithm may also play an important role in achieving good executing speed. The GC steals valuable CPU time from the main application that is running.

\(^1\) There are other reasons behind these problems as well, but these have a smaller effect on overall performance and will not be discussed in this report.
2.4.2 Execution smoothness

Execution smoothness can be described as applications appearing to be running smoothly. Movies should be played at a constant frame rate and not pause for a second or two in the middle of the movie. Games should also be played at a constant frame rate and not vary too much. See chapter 3 for more details about smoothness and how it’s measured. Execution smoothness is a big problem in real-time applications written in Java due to two major technologies: garbage collection and JIT compilation. It’s very subjective what one may think is smooth enough, but nonetheless, smoothness is desired.

Garbage collection

The first issue is the garbage collector (GC). When the GC is running, it steals valuable CPU time from the main application. This results in short pauses in the execution, and to the user the application may appear to stutter.

The programmer can affect the GC in at least one way. If there is no garbage to collect, the GC will not use much CPU time. Then the programmer will have to use dirty tricks, for instance use a pool of objects so that they are reused rather than to create new objects. This is no good workaround since it will make the code less readable and more error prone.

A better solution in the long run is to write clean and readable code, and solve the problem with smoothness using some other technique, for instance a hardware accelerated Java virtual machine or perhaps better GC algorithms.

Compilation

The second technology which decreases smoothness is compiler based JVMs. When the compiler is compiling code, it steals CPU time from the main application just as the GC does.

A hardware accelerated JVM might improve smoothness because bytecodes are interpreted faster. This could also be a problem in case the JVM decides to compile more code since it’s faster, but this is configurable.

2.4.3 Memory usage

If a compiler based JVM is used, memory usage is increased. When the compiler compiles bytecodes into native instructions, the bytecode will take at least four times more memory. The reason for this is that bytecodes are 8 bits long and native instructions usually are 32 bits long\(^2\). If more than one native instruction is needed, the compiled code will use a multiple of four bytes instead.

\(^2\)This depends on the architecture of the CPU.
2.4. Areas of interest

Compiler based JVMs increase the memory footprint and memory can be a very limited resource in some environments.

2.4.4 Startup time

Startup time is slow due to that a lot of classes are loaded when the JVM is started. If a compiler based JVM is used, it may decide that some of the classes loaded at startup also needs to be compiled. This may result in the application running faster once it’s started, but it also increases startup time even further. This problem can however be solved with AOT, but will then require more storage space instead.

If hardware acceleration is used, bytecodes are executed faster which may result in decreased startup time.
Chapter 3

Methods to measure execution smoothness

In this chapter, the concept of smoothness and how it relates to execution smoothness is defined. The concept of smoothness over different periods of time is discussed and so are the problems it involves. This results in a proposed model for measuring smoothness. Finally, the model is verified practically using synthetic, well-controlled tests, and by conducting a survey to make sure the results coincides with visually experienced smoothness.

3.1 Definition of smoothness

![Graph 1](image1.png) ![Graph 2](image2.png)

Figure 3.1: An example of a rough graph both locally and globally. This is a realistic, but not desired.

Execution smoothness, or in other words how the frame rate varies over time in games and other real-time applications, can be measured in different time spaces. Locally, i.e. in a short period of time, big variations are considered not to be smooth. An
Figure 3.2: An example of a smooth graph both locally and globally. This is the desired case, but isn’t very realistic.

Figure 3.3: An example of a graph which is rough globally, but is smooth locally. This is a more realistic and also desired graph.

example is a signal with high frequency noise applied to it as can be seen in the graph to the right in figure 3.1. Globally, for a long period of time that is, those variations may not be noticeable. Instead a big global variation may be considered not to be smooth. An example is a sinus wave with a low frequency but with a high amplitude, see the left graph in figure 3.1. For reference, an example of a graph which is considered to be smooth both locally and globally can be seen in figure 3.2. This is the desired behavior, but isn’t very realistic since the rendered image isn’t static — for instance, more enemies on the screen means more sprites or polygons to draw, and the JVM cannot affect that. A more realistic behavior is a mixture where it may be rough globally and smooth locally, and this is also a desired behavior, refer to figure 3.3.
3.2 Measuring smoothness

In this chapter, an existing model for measuring smoothness is examined, and a new model is proposed that works in the cases where the first model fails.

3.2.1 Existing models

There is no de-facto model for measuring smoothness in this domain (how frame rate in games varies over time). It is, however, possible to borrow ideas from other domains such as mathematics and signal theory where noise is an important issue.

Modulus of continuity

One popular way of measuring smoothness of a function in mathematics, is to use the modulus of continuity [8]. The definition is as follows:

Let \( f \) be a function with real values. Let \( t \) be some point and let \( \delta \) be a positive number. We define the local modulus of continuity at the point \( t \) by

\[
\omega_f(\delta; t) = \max_{s:|s-t|<\delta} |f(t) - f(s)|.
\]

The modulus of continuity (sometimes referred to as the global modulus of continuity) is defined by

\[
\omega_f(\delta) = \max_t \omega_f(\delta; t).
\]

This will work very well for graphs which are very smooth locally. However, it will not work well for some graphs. For example, for \( f = \sin(x^2) \) it’s easy to see that \( \omega_f(\delta) = 2 \) for any \( \delta \), even though this is a nice smooth graph. This model will therefore not be investigated further.

In the next section, a new model is proposed and discussed.

3.2.2 Our model

To solve the problems with globally rough graphs, the graph has to follow the other graph globally, but not locally. In other words, the noise should be removed. This can be accomplished by convolving the original function \( f \) with a kernel \( h \) (a low pass filter, see equation 3.5). A smoother curve is then obtained, called \( \tilde{f} \), see figure 3.4.

The order \( k \) of the filter can be fine tuned. By calculating the root mean square (RMS, also sometimes referred to as the quadratic mean), a value describing the
smoothness is obtained. The lower the value, the smoother graph is considered to be. See equation 3.3 for the mathematical definition.

\[ S = \sqrt{\frac{\sum_{t=0}^{n} (f(t) - \hat{f}(t))^2}{n}} \]  \hspace{1cm} (3.3)

\[ \hat{f} = h^k \ast f \]  \hspace{1cm} (3.4)

\[ h = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \] \hspace{1cm} (3.5)

Different kernels were tested (for instance an averaging filter with variable window size, \( \frac{1}{w} \begin{bmatrix} 1_1 & 1_2 & \ldots & 1_w \end{bmatrix} \)), but the one in equation 3.5 worked the best. A value of 64 was verified empirically to be a good value of \( k \). Convolution can be performed in different ways on the border values, for instance using replication or reflection. Array values outside the bounds of the array were computed by reflecting the array across the array border.

This model works well for both of the two synthetic test cases, refer to table 3.1 and figure 3.5.
Figure 3.5: Data set 1 can be seen in the left column and data set 2 in the right column. The smoothness values can be seen in the figures. The lower the value, the smoother the graph is considered to be. These graphs are synthetic with simulated GC activity around $t = 50$, $t = 100$ and $t = 150$. 
**Data set 1 (synthetic)**
plot 1 8.39
plot 2 4.33
plot 3 2.51

**Data set 2 (synthetic)**
plot 1 8.54
plot 2 4.27
plot 3 2.13

*Table 3.1: Results from our model on two data sets.*
3.3 Verifying the model – a survey

Smoothness is a very subjective property. To verify that the method to measure execution smoothness coincides with experienced smoothness, a survey was conducted. The test consists of some balls bouncing around on the screen (see figure 3.6).

![Screenshot from the test MIDlet used to verify the smoothness model.](image)

**Figure 3.6:** Screenshot from the test MIDlet used to verify the smoothness model.

Two games were simulated; one running at a base framerate at 50 FPS and one running at 30 FPS. DAC and GC pauses were then simulated by using `Thread.sleep(t)` to lower the FPS to desired values:

```java
while (true) {
    long currentTime = System.currentTimeMillis();
    int dt = (int) (currentTime - lastUpdate);
    lastUpdate = currentTime;

    // ... other code (move objects, update graphics etc)

    // limit fps and simulate DAC/GC pauses
    long timeToSleep = 0;
    if (currentTime - lastPauseTime > PAUSE_INTERVAL) {
        lastPauseTime = currentTime;
        timeToSleep = pauseSleepTime - dt;
    } else {
        timeToSleep = baseSleepTime - dt;
    }
    Thread.sleep(timeToSleep);
}
```

The survey was conducted with 17 people. They ran the different simulations and were also asked to rank the tests in order of smoothnesses. The smoothness was also measured by the method described in the previous section.
Table 3.2: Survey results to verify that the model for measuring smoothness coincides with experienced smoothness. The rank shows how smooth the test was experienced visually (1 — the smoothest) and the smoothness score shows the calculated smoothness (low score — smooth). The vote accuracy shows how many percent of the participants that gave a test the specific rank.

The results of the survey and the calculated smoothness scores can be seen in table 3.2. The results show that the model for measuring smoothness works well and coincides with experienced smoothness and that it can be helpful if it’s hard to decide visually (see the survey results for the test with base FPS at 30).
Chapter 4

Testing

This thesis evaluates how a hardware accelerated JVM performs compared to a JVM using other execution techniques. Focus in the testing were on the four areas; execution speed, execution smoothness, startup time, and memory usage. For testing, both real world MIDlets, such as games, and pure benchmarking MIDlets were used.

4.1 Test configurations

To investigate how hardware acceleration performs compared to different execution techniques, the following configurations were used:

- **Interpreting mode** – Each byte code is interpreted and mapped to native instructions.
- **Compiler mode** – A block of byte codes is compiled into native instructions.
- **Hardware accelerating mode with interpreting** – Byte codes are interpreted directly by the processor.
- **Hardware accelerating mode with compiler** – Both compiler and hardware acceleration are used.

To investigate how the Java heap influence the performance of a particular execution engine, all of them are tested with different Java heap sizes; 512 kb, 1 Mb, 1.5 Mb, 2 Mb, and 3 Mb. An execution engine with a specific heap size is henceforth called a configuration.
4.2 Testing methods

All tests were run three times with each configuration. An average of the score reported in each test run was calculated. This was done since there is a small variation in the results due to tasks running in the background and variation in game play. The variation is almost impossible to foresee. If the result in a test differed very much from the other results, this result was discarded and the test was repeated. The phone was restarted when testing a new configuration but not between the different test MIDlets.

From the pure benchmarking MIDlets a number is reported, representing the result for all the tests. Since the scores themselves aren't very interesting, but rather the difference, the results are normalized with the result for a base configuration (1 Mb of heap memory, JIT compiler off, and hardware acceleration off). This makes it easier to compare the results.

4.2.1 Benchmarking execution speed

To benchmark execution speed, both pure benchmarking MIDlets and games were used. The benchmarking MIDlets Amark 1.3 (see section 4.3.1), JBenchmark 2 (see section 4.3.2), and Grinderbench\(^1\) (see section 4.3.3) were used. The values reported from the benchmarks are treated as a measurement of execution speed.

The games Racing Fever 2 (see section 4.3.4), Racing Fever 3D (see section 4.3.6), Extreme Air Snowboarding (see section 4.3.5), and 3D Extreme Air Snowboarding (see section 4.3.7) were used.

The FPS value is treated as a value of the execution speed. High FPS values implies high execution speed.

4.2.2 Benchmarking startup time

To benchmark startup time, the time was measured from the starting of the MIDlet in the JVM until the MIDlet paints on the display for the first time. Startup time was measured like this because the MIDlet is experienced by the user to be started first when it paints on the display.

Startup time was measured for the games Racing Fever 2, Racing Fever 3D, Extreme Air Snowboarding, and 3D Extreme Air Snowboarding.

The resolution of the JVM timer isn't exactly 1 millisecond, but differs from different JVM implementations. This introduces a little source of error, but the error is neglectable.

\(^1\)Due to licensing issues these tests were executed by Ericsson AB on our behalf
4.3. Test applications

4.2.3 Benchmarking execution smoothness

By measuring the FPS as described earlier in this chapter and using our model for measuring smoothness, as described in section 3.2.2, a number representing the smoothness were obtained.

The games Racing Fever 2, Racing Fever 3D, Extreme Air Snowboarding, and 3D Extreme Air Snowboarding were used.

The FPS value was logged four times per second. This introduces a small error. It would have been better to log each screen refresh, but the overhead for this would have been too big. This means that the logs are a bit less detailed, but this doesn’t make a big difference because all tests have the same conditions.

4.2.4 Benchmarking memory usage

To measure memory usage, the heap space allocated by the MIDlet was recorded. The memory usage is defined to be the total amount of memory used by the MIDlet during the test.

Memory usage was recorded for the games Racing Fever 2, Racing Fever 3D, Extreme Air Snowboarding, and 3D Extreme Air Snowboarding.

4.3 Test applications

The MIDlets that are used in this thesis are described in the following sections. They were picked together with Ericsson, and found to be most suitable for this evaluation.

4.3.1 Amark 1.3

Amark 1.3 [2] is a very commonly used benchmark MIDlet. It was run in full screen. The outputs from the tests are either a score or a time in milli seconds. The benchmark consists of the following tests:

- **startup** – This test move characters and simple shapes around on the screen.
- **3d-lines** – This test rotates wire frames cubes on the screen.
- **3d-stars** – This test move many single points on the screen to simulate stars.
- **shapes** – Fills shapes using Midp 1.0 drawLine().
- **flag** – Moves an image on the screen in a wavy pattern.
- **100-triangles** – Draws 100 filled triangles on the screen and measures the time.
• chunky1 – Heavily tests drawLine() functions with many small lines (one line per pixel) by performing a chunky to image conversion and measures the time.

• chunky2 – Heavily tests fillRect() functions with many small images (one per pixel) by performing a chunky to image conversion and measures the time.

• fractal – Performs fractal operations on an image.

• mosaic – Makes an image disappear using clipping.

• zoomer – Scales an image in real time.

• tmap – Tests performance of real time texture mapping on a spinning cube.

• voxel – Generates and moves a voxel landscape.

4.3.2 JBenchmark 2

JBenchmark 2 [7] measures the graphical performance of Java enabled mobile devices which support the MIDP 2.0 specification. The outputs of the tests are a value calculated by JBenchmark 2. The benchmark consists of the following five tests:

• Image manipulation – Scales an image and displays the result in real time.

• Text – Scrolls a long text.

• Game – A scrolling platform game in 2D.

• 3D transformations with shading – Rotates a chess piece on the screen.

• User interface – Displays and moves around various UI components.

4.3.3 EEMBC GrinderBench

Embedded Microprocessor Benchmark Consortium (EEMBC) has developed GrinderBench [3]. This benchmark only test CLDC and doesn’t need any implementations from additional JSRs. GrinderBench doesn’t contain any graphical test at all. The benchmark consists of five tests:

• Chess – A complete chess playing engine that is used to determine a set of chess moves. The chess benchmark only performs the logical parts of a chess program, as no graphical output is available. It plays a number of games with itself and times how long it takes.

• Crypto – This suite of algorithms measures the performance of Java implementations in cryptographic transactions, such as those used when goods and services are paid for via a mobile device. The DES, DESede, IDEA, Blowfish, and Twofish encryption algorithms are exercised.
4.3. Test applications

- **Kxml** – Measures XML parsing and DOM tree manipulation.

- **Parallel** – Exercises a Java implementation’s ability to perform its user interface while interacting with the Internet, particularly in scenarios where applications are divided into separate application threads with some communications threads running in the background behind the thread providing

- **PNG decoding** – Shows how fast a Java implementation can decode a PNG photo image of a typical size used on a mobile phone.

4.3.4 Racing Fever 2

![Racing Fever 2](image)

*Figure 4.1: Racing Fever 2*

Racing Fever 2 [2] is a racing game in 2D and uses sprites (ordinary images) heavily. Most games are still in 2D.

4.3.5 Extreme Air Snowboarding

Extreme Air Snowboarding [5] is a snowboard game in 3D. Modern games are moving toward 3D as processors are getting faster. It uses its own 3D engine (not everything is “pure” 3D, it uses sprites as well) and not the 3D API Mobile 3D Graphics API for J2ME [28].

4.3.6 Racing Fever 3D

Racing Fever 3D is an updated version of Racing Fever 2. The MIDlet uses Mobile 3D Graphics API for J2ME [28].
4.3.7 3D Extreme Air Snowboarding

3D Extreme Air Snowboarding [6] is an updated version of Extreme Air Snowboarding that uses Mobile 3D Graphics API for J2ME (see [28]).
Figure 4.4: 3D Extreme Air Snowboarding
Chapter 5

Test results and discussion

In this chapter results of the measurements for each area of interest are presented and discussed.

5.1 Execution speed

![Execution speed measurements for the benchmarking MIDlet JBenchmark 2 (normalized with base configuration: 1Mb heap, DAC off, HW acc off).](image)

The DAC does a good job when it comes to execution speed – both in benchmarking MIDlets (see figure 5.1) and in games (see figure 5.2). When the heap size is increased, performance is also increased since the DAC can compile more methods and store the compiled code in memory. When the heap size is small, there isn’t much memory available for the DAC to store compiled code in. This means in practice that the DAC will fail to compile a lot of methods and not do much good.
Using the DAC in conjunction with hardware acceleration gives the best performance in most tests, but not all. Combining both techniques is not always as good since some bytecodes may then be executed in software instead of hardware (if they aren’t supported by the hardware accelerator and aren’t compiled).

The hardware accelerated VM performs about the same as the DAC enabled VM, at least in real world applications such as games. In some benchmarking programs with tight loops which the DAC detects as frequently run code segments and compiles into native code, the DAC outperforms the hardware accelerated VM. All of the GrinderBench\(^1\) benchmarking tests are for instance very DAC friendly (see figure 5.3). A hardware accelerated VM doesn’t have the same problem as a DAC enabled VM.

\(^1\)Due to licensing issues these tests were executed by Ericsson AB on our behalf
when the heap size is small (see figure 5.1), but instead performs just as well across all heap sizes.

A pure interpreting VM is always slower than the other configurations described above.

If the heap size is big enough for the DAC to do its job, a DAC enabled VM is the best configuration to use. However, if the heap is small, a hardware accelerated VM is a better choice.

## 5.2 Startup time

Figure 5.4 shows the average startup times for the games that were tested. The startup times for the MIDlets were added together for each configuration and then divided by the number of tests to obtain an average number.

![Startup time graph](image)

**Figure 5.4:** Average startup times for the MIDlets 3D Extreme Air Snowboarding, Racing Fever 3D, Extreme Air Snowboarding and Racing Fever 2 (normalized with base configuration: 1Mb heap, DAC off, HW acc off).

The startup time is long when the DAC is enabled – it is increased by about 5-20% in average as can be seen in figure 5.4. The reason for this behavior is that the DAC compiles a lot of methods and thus stealing valuable CPU time. When the heap size is increased, startup time is decreased. The reason for this is that the heap doesn’t get filled with compiled code as fast, which in practice means that the GC doesn’t have to run until the MIDlet is up and running. Running with the DAC and the hardware accelerator at the same time decreases startup time slightly.

For the hardware accelerated VM, startup time can be decreased by about 10-20% in average. The reason for this is that bytecodes are simply executed faster directly in hardware than in software.

A pure interpreting VM is not bad at all. It’s faster than the DAC enabled configuration, and doesn’t fall far behind the hardware accelerated configuration.
The best configuration to use, regarding startup time, is a hardware accelerated VM or a pure interpreting VM. A big heap is preferable for all configurations, but if a DAC enabled VM is used, a big heap is more or less a requirement to guarantee the heap isn’t filled with compiled code too fast. A solution to improve startup time with the DAC enabled without increasing the heap, is to precompile the MIDlets prior to execution (at installation for instance), also called AOT (see section 2.2.1).

5.3 Execution smoothness

Figure 5.5 shows typical results regarding execution smoothness.

![Execution smoothness](image)

**Figure 5.5:** Execution smoothness measurements for the game Extreme Air Snowboarding (normalized with base configuration: 1Mb heap, DAC off, HW acc off).

The DAC enabled configuration generates a very rough graph (see figure 5.6). When the DAC is enabled, it sometimes steals CPU time from the running MIDlet. More garbage is also created when the DAC is enabled, which means that the GC must run more frequently or collect more garbage when it runs.

Running with the DAC in conjunction with hardware acceleration, smoothness is worsened a little bit compared to the solely DAC enabled configuration. The reason for this is that hardware acceleration frees up more memory (as can be seen in section 5.4), which in turn gives the compiler a chance to run even more frequently.

Disabling the DAC is the most effective way to improve smoothness. Running in interpreted mode is very smooth, pretty much comparable to using the hardware accelerator. Enabling the hardware accelerator is the best way to achieve smooth execution, while still offering much better execution speed than interpreted mode.
5.3. Execution smoothness

Figure 5.6: Comparison between different VM configurations for the game Racing Fever 2. From top left to bottom right; interpreting mode, DAC enabled, hardware acceleration, DAC enabled and hardware acceleration.
5.4 Memory usage

Figure 5.7 shows typical results regarding memory usage.

![Memory usage (3D Extreme Air Snowboarding)](image)

**Figure 5.7:** Memory usage measurements for the game 3D Extreme Air Snowboarding (normalized with base configuration: 1Mb heap, DAC off, HW acc off).

The results for memory usage can be a bit misleading at a first glance. The results are only comparable within each heap configuration, in other words there is no point in comparing the results for the configuration with 512 kb of heap memory with the one with 3 Mb. The VM will simply use more memory if more memory is available – there is no need to be conservative if there is a lot of available memory, at least not until it is needed by someone else.

Running with the DAC enabled will always use the most memory. It’s important to remember that it doesn’t require more memory, it will only use more memory if more memory is available. This means that if there isn’t a lot of heap memory available, the DAC won’t succeed in compiling any methods. Using the hardware accelerator in conjunction with the DAC doesn’t affect memory usage much.

Turning off the DAC will use the least amount of memory. Enabling the hardware accelerator increases the memory consumption slightly, but not as much as the DAC does. The hardware support code needs a little bit of memory to be able to translate bytecodes into instructions supported by the hardware.
Chapter 6

Conclusions and further work

Java in embedded devices has become more and more popular. MIDlets are getting more advanced and many new Java APIs for J2ME are developed. MIDlets need to run fast, smooth, require small amount of memory, and have a short startup time. These demands need to be fulfilled by the JVM. The JVM can use several techniques to improve one or a number of these demands. Techniques that are examined in this thesis are JIT compilation, interpretation, and hardware acceleration. The technique that is used often depends on the constraints of the environment, one technique may be the best for one configuration but not for another.

6.1 Conclusions

Java has many desirable features, but has performance problems in some areas. The focus was on the areas execution speed, executions smoothness, startup time, and memory usage.

To be able to see what impact hardware acceleration had on the areas described above and to see how it performed compared to other execution techniques, the following configurations were used:

- **Interpreting mode** – Each bytecode is interpreted and mapped to native instructions.
- **Compiler mode** – A block of bytecodes is compiled into native instructions.
- **Hardware acceleration with interpreting mode** – Bytecodes are interpreted directly by the processor.
- **Hardware accelerating with compiler mode** – Both compiler and hardware acceleration is used.
Mobile phones have different memory configurations, high-end phones may have quite a big Java heap while low-end phones may have a very small one. All the the areas were tested with different heap configurations ranging from 512 kb to 3 Mb of heap memory.

Since there was no standard for measuring execution smoothness, a model to measure execution smoothness was also constructed. The model consists of creating two graphs; one consisting of the measured FPS and one which follows the first one globally, but not locally, in other words, the noise is removed. A smoother curve is then obtained, called $f$. The order $k$ of the filter can be fine tuned. A value of 64 was verified empirically to be a good value of $k$.

The model proved to work very well by visually comparing graphs and the obtained smoothness values. However, a 50\% decrease in smoothness value doesn’t necessary mean the graph is twice as smooth – it just means the graph is smoother – good for comparing different configurations to see which configuration is the best.

Tests shows that all of the different configurations have their own strengths and weaknesses. Good execution speed can be obtained with both the DAC and by using hardware acceleration. Short startup times can be obtained by turning off the DAC or by increasing the heap if the DAC is enabled. To achieve smooth execution, it’s best to turn off the DAC. To cut down on memory usage, an interpreting VM is the best choice. From a device manufacturer point of view, it’s very important to achieve good scores in benchmarking programs.

Below is a condensed list of pros and cons.

**DAC enabled VM**
+ good execution speed in games
+ good execution speed in benchmarking programs
+ doesn’t cost anything extra
  - long startup times
  - non-smooth execution
  - requires more heap to work well, which costs money

**Hardware accelerated VM**
+ short startup times
+ doesn’t require much heap
+ good execution speed in games
± average to good execution speed in benchmarking programs
± pretty smooth execution
  - costs money

**DAC in combination with hardware acceleration**
+ good execution speed in games (faster than DAC only in some games)
+ good execution speed in benchmarking programs
  - long startup times
  - non-smooth execution (worse than DAC only)
  - requires more heap to work well, which costs money
  - hardware acceleration costs money
Interpreting VM
+ short startup times
+ smooth execution
+ doesn’t require much heap
+ doesn’t cost anything extra
- bad execution speed in games
- bad execution speed in benchmarking programs

For a low-end device with limited resources, hardware acceleration is the best choice. For a high-end device with lots of RAM and a very fast CPU, a DAC enabled JVM is the best choice.

In other words, it’s a compromise.

6.2 Further work

This thesis looks at many aspects of accelerating Java in hardware, but leaves room for more work.

6.2.1 Other hardware accelerators

It would be interesting to test other hardware accelerators. They are probably designed differently and may be better at some areas and worse in some.

6.2.2 Other Java virtual machines

It would also be interesting to test other JVMs in conjunction with different hardware accelerators. One particular hardware accelerator may work better with one particular JVM compared to another.

6.2.3 Other domains

The work in this thesis is limited to the scope of mobile phones and Ericsson’s platform. An investigation of the possibilities to add hardware acceleration into future CPUs from chip manufacturers such as Intel or AMD would be very interesting. In that case perhaps the difference in performance between applications written in C/C++ and Java may be further minimized. A DAC enabled JVM performs very well on desktops though, mostly due to access to a lot more RAM and faster CPUs, so the difference may not be so big, but is still worth investigating.
6.2.4 Power consumption

If a software interpreter loop is used, one bytecode may result in several native instructions, as described in section 2.4.3. If hardware acceleration is used instead, fewer native instructions are used. This would in theory mean fewer clock cycles, and hence power consumption should be lowered\(^1\). This is very important in resource constrained systems such as mobile phones. This is however not investigated in this thesis, but is nonetheless interesting.

\(^1\)This probably only applies to hardware technologies where the accelerator is integrated into the main CPU.
Bibliography


