MASTER'S THESIS | LUND UNIVERSITY 2016

Investigation of High-Level Language Support in a Resource-Constrained Embedded Environment

Philip Mårtensson, Daniel Olsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-12

Investigation of High-Level Language Support in a Resource-Constrained Embedded Environment

Philip Mårtensson
ada10pma@student.lu.se

Daniel Olsson
atf10dol@student.lu.se

May 2, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Assad Obaid, Assad.Obaid@axis.com
Patrik Persson, patrik.persson@cs.lth.se

Examiner: Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se

Abstract

Personal computers have gained a significant boost in computational power and digital storage space at a reduced cost in the last decade. In the search of increased programmer productivity and cross platform portability, language popularity has shifted from lower level languages such as C to higher level languages such as Java and C#. Many of today's embedded systems are experiencing the same development as the personal computers did. However, most companies dealing with embedded devices still use C. We investigated what effect a shift like this would have at Axis Communications. The study was done by setting up C# and Java on a camera and conducting performance tests on it. The analysis showed that when using C# as a replacement for C, we saw improvements in programmer productivity whilst still upholding performance for some applications. For the most performance intense use cases, the performance requirements were not satisfied. With the growth of high-level languages, we do see a bright future for the support for them in embedded systems.

Keywords: high-level programming languages, resource-constrained, profiling, embedded systems, cross-compiling

Acknowledgements

We would like to give special thanks to our supervisor at Axis Communications, Assad Obaid, who gave us much of his valuable time in order to provide us with technical details about the hardware and software used at the company. He was always enthusiastic about our work and provided us with internal contacts and was always available for guidance and advice.

We would also like to express great gratitude to our supervisor at Lund University, Patrik Persson, who gave us extremely helpful guidance with thesis from start to end. Thanks to him we never lost track of what we were doing. He showed great enthusiasm and provided us with vital feedback and many interesting conversations. Without him, it is hard to say if we would have completed the work within our timeframe.

Finally, we would like to thank Axis Communications and all the employees we met there that took an interest into the work we did and provided us with technical assistance and all the tools we needed to carry out the work.

Contents

Chapter 1
Introduction

C for performance and low level? Python for simplicity? JavaScript for popularity? C# for elegant flexibility? It is not always easy choosing a development platform for an embedded system that should be easy to program, efficient and last a long time. Especially not in the time of a shift where hardware is becoming cheaper and software is becoming better. Because of this, embedded systems are experiencing a similar situation as the general purpose computers did a few years ago when high-level languages saw a great increase in popularity. We will refer to C as a low-level programming language and C# and Java as high-level programming languages.

Axis Communications is one of the companies that want to explore the feasibility of running higher level languages in their resource-constrained embedded devices. Their motive behind this investigation is that they still use C to write high-level applications for all of their cameras, even though the cameras have become a lot more powerful since they started developing in C.

We will investigate the following:

> Have modern embedded devices, such as cameras at Axis Communications, reached a point where it is feasible and advantageous to transition to higher level languages instead of C?

1.1 Background

Traditionally, embedded systems have required resource-efficient environments [1]. The reason for this has been the requirements on size and cost. Because of this, the tools that the systems have used have had to be efficient with as little overhead as possible. Due to the lowered cost on more powerful hardware, it is now interesting to investigate if it could be feasible for embedded systems developers to switch to a platform on a higher level in order to gain efficiency and stability and possibly a shorter development cycle.

M. Nahas and A. Maaita mention important considerations when it comes to choosing programming language [1]:

> ”In real-time embedded systems development, the choice of programming language is an important design consideration since it plays a significant role in reducing the total development time.”

Many embedded devices are designed to be used as real-time systems, meaning that the systems might have hard deadlines to meet and thus require the devices to have sufficient performance, schedulability and determinism [2]. High-level languages are still generally considered to be unsuitable for these real-time embedded systems [3]. This is due to factors such as the lower performance which can make the machines miss important deadlines and also due to the fact that garbage collectors often are present and mess up important timing by being invoked at unpredictable times. However, embedded systems or parts of them that do not rely strictly on real-time requirements are also common but not as extensively studied as their more demanding counterparts. The less strict requirements makes it interesting to investigate the actual *feasibility* of migrating to a higher level language on modern resource-constrained devices, by using a trade-off between performance and programmer productivity. For example, in Java, there is the *RTSJ*, real-time Java specification. It is designed to support both hard and soft real-time applications [4]. See section 2.6 for differences between hard and soft real-time.

The motivation behind the work is that Axis Communications today uses the programming language C when implementing high-level applications on their camera platform. They are not alone when it comes to this choice, a study in 2006 found that over 50% of embedded system developers used C for the implementation of their projects [1]. Axis hardware is becoming more and more powerful and therefore we want to investigate whether it is possible to change programming platform. This could mean an improved productivity for the sole developer because of large open source and rich class libraries, improved debugging and testing possibilities, large programming communities and also more technical advantages such as garbage collection and object orientation.

In our group at Axis Communications, opinions have been expressed regarding the hardships and problems when it comes to work efficiency. There is a strong will to explore the possibilities of trying a high-level language for their work.

The goal is to investigate whether it is possible to transition, from C, partly or completely to a high-level language such as C# or Java without losing significant performance that would interfere with the quality of the product. It is important to do this investigation on the embedded device in question, as we need to consider the architecture, operating system and other device-specific factors to reach a viable conclusion.

The end result contains the results of the measurable performance difference for the investigated frameworks as well as the softer issues regarding developer efficiency. These measurable results are evaluated and compared to corresponding applications written in C. The purpose of the evaluations is to give an answer to the question about whether a transition is feasible for all applications, only for some, or not at all. It is hard to set a specific limit where high level languages stops being feasible on an embedded device, one needs to consider the upper limit for each program in all situations it will be used. Therefore, we will consider it feasible if the tasks executes within a good time frame and uses a reasonable amount of resources for the task in question. It is evaluated on a case by case basis whether the measured results are good or bad.

For the work carried out in this thesis, the interest lies mainly with previous research on the performance of high-level languages in embedded devices having limited hardware resources. A lot of study has been made for high-level language support for real-time embedded systems, often by using Java [2]. While real-time embedded systems take more aspects into consideration than just performance, it is still very important for those systems, as they need to be able to meet scheduling deadlines [5]. So while real-time embedded systems are not the primary topic of discussion for this thesis, a lot of the research carried out on performance factors in real-time embedded systems is of high relevance for this work. This is because our work can be considered as a subset of the work done for real-time embedded systems, but with less requirements on schedulability and determinism as is required by hard real-time systems. This subset is called soft real-time systems and often considers throughput (e.g. the number of frames per second a camera handles) instead of strict deadlines. In soft real-time systems it does not matter if a deadline is missed every now and then.

1.2 Previous Work

Performance of programming languages is an area that has been a subject of much research. Both execution performance and programmer efficiency has been a topic of interest for many different languages. The languages chosen to be explored in this thesis, C# and Java, are very popular languages and has of course been included many times in this type of research [6].

1.2.1 C# in Real-Time Embedded Systems

Michael H. Lutz, Phillip A. Laplante discuss the usability of the .NET Framework and C# in embedded real-time systems in [2]. They mention the many aspects of the C# language that makes it an interesting language to run on embedded systems. An example of this is the ability to access physical memory locations by using *unsafe code*. Having this feature will allow the developers to communicate with the different hardware components available in the device, which most likely is a very important feature in case a full transition ever is made from C to C#. For most applications, such as the ones tested in this report, physical memory access is not very important. Components on a lower level, such as drivers, will still be written in C due to reasons such as not needing any virtual machine in order to run.

Michael H. Lutz and Phillip A. Laplante conducted execution performance tests and measurements of the memory footprint required by the C# in a nowadays weak computer and compared it with native C code [2]. While it is difficult to compare the architecture of this computer with the camera we investigate at Axis Communications, we can still say that they are not too far from each other in terms of performance. The CPU of the computer in the article has a higher clock frequency and a few hundreds of megabytes more RAM, but the CPU architecture in the camera is more modern. More details about the camera hardware we used is found in section 3.1.

The measurements in the article were made on a system running Windows with the first available non-beta version of the .NET framework. This is somewhat different from the tests performed in this thesis. Aside from the differences in the system hardware, the platform we tested C# and the .NET Framework on is a Linux-based platform and runs an open source alternative implementation of the .NET framework instead of the official implementation by Microsoft. Another difference is that the tests done the article only test the execution performance of C# versus C using rather primitive benchmarking programs. These programs measured the number of floating point operations done over time, as well as allocation and deallocation of linked lists for both of the languages. Our work focuses more on trying to simulate actual use-cases suitable for the company that develops the embedded systems we run our tests on. By doing this we get a better overview of how the performance looks in actual practical cases instead of just pure number crunching situations were we know native code will almost always outperform the virtual machines due to less overhead. Finally, another important difference between our tests and the ones performed in the article is that we are implementing the applications in such a way that a programmer would have written the program in that language. In the article they wrote the original program in C# and then copied the program to another file and changed it so that it would compile with a C compiler. The authors argued that this would keep the programs as comparable as possible. We want our programs to be as practically realistic as possible instead, that is we want to make use of available language idioms and suitable features, while still solving the same problem and thus allow us to compare them.

The results from the article showed that C# performed very well in the floating point test, having very similar performance as the C version. The main performance issue that the article showed was that the memory management done by the garbage collector was outperformed by the manual memory management done in C. This is a very important result for our purposes, as the memory is going to be shared with multiple other programs running in parallel on the embedded devices we are testing. Too high memory consumption might make programs written in C# unfeasible even if they perform well in terms of execution speed.

The article ends the discussion by stating that C# was not ready for hard real-time embedded systems at the time of its writing. This was due to the lack of important threading constructs and other minor issues. It does however state that further study is required in order to reach the same conclusion for soft and firm real-time systems. The systems we are studying in this report is close in definition to that of soft real-time systems, since we are mostly concerned about the performance and not so much about missing deadlines.

Mouaaz Nahas and Adi Maaita states that the .NET family languages had gained popularity in the field of embedded systems development [1]. However, they also stated that due to the fact that .NET family languages are object oriented, they were not considered the best choice since it imposes too much overhead. It is important to note that this article is at the time of this writing 10 years old, and it can be considered relevant to look at this problem once again.

1.2.2 Comparison of Programming Languages

In order to be able to perform measurements for the productivity of programmers in different languages, an evaluation process needs to be well defined. In an article by Lutz

Prechelt, he makes an empirical comparison of seven different languages, including both Java and C. The comparison takes program length, programming effort, runtime efficiency, memory consumption and reliability into account. Many of these aspects are the same as the ones we take into account for our investigation [7]. However, we also look at the size on secondary memory due to its high relevance in resource-constrained environments.

Prechelt also performs a programmer efficiency test by measuring the lines of code (LOC) written by multiple programmers implementing the same program in different languages. However, measuring the productivity in LOC is often considered as a bad metric [8]. Although LOC still says something about productivity if there is a significantly large difference in it between two different language implementations. The article also performs programmer efficiency tests by looking at the total working time that was needed for the programmers to implement the programs in question. Furthermore, the article looks at the design decisions made by the programmers in the different languages. Most of the key differences found concerning the design decision were between scripting languages and the statically compiled languages. It also found that the design and implementation of the test program took significantly less time when done in the scripting languages. In our investigation scripting languages are not considered at all. This is partly due to the lack of time to do so and partly because scripting languages were initially thought to be too ineffective for resource-constrained environments due to being mostly interpreted languages [9].

Even though Prechelt does not perform the tests on resource-constrained embedded devices, the result of the comparison is still valid for our work if only the difference between the languages are considered, and not if the performance is practically feasible in the tested environment. Especially the results regarding programmer efficiency can be considered important, as they are completely irrelevant to the machine running the code.

The test programs in Prechelt are implemented by different individuals, all in their own way. The resulting programs thus look and perform very different even though they solve the same problem. In our work we perform similar performance tests to this, since we also base our work on implementations done ”in such a way the programmer would have coded it if they used that language”, and we also consider if the performance differences are practically feasible in the resource-constrained embedded environment in question. One important thing to note is that due to the age of the discussed article, the Java version that was investigated was very old (Sun JDK 1.2). The Java versions available today have improved vastly in terms of performance and features and has seen many optimizations [10].

As the authors of the web site *The Computer Language Benchmarks Game* suggest, a program’s execution time does not only depend on the environment, but also how it is written [11]. They also suggest that in order to really evaluate a programming language and its environment whilst making it fair, one has to implement a big system. The reason for using ”toy benchmark programs” as they call it, is simply because there is not enough time to make an experiment with programs that consists of 10000+ lines of code. Luckily, there is still some value to using toy programs, you can still learn from it [12].

1.3 Limitations

There are countless of platforms, environments and programming languages. We will not investigate them all. We chose Java and C# because they are two of the five most popular general-purpose enterprise programming languages of 2015 [6]. Google Go was interesting but we chose to exclude it due to its immature state in embedded programming [13]. Other languages such as C++, Python, Ruby or Scala were not investigated. C++ was an interesting contender, but since it is very similar to C, we chose to not go with it.

The platform running C# is Mono and Oracle's Runtime Environment for Java. There may be other runtime environments that could give different results.

It is possible to compile high-level languages to machine code and therefore skip the virtual machine, but we felt that it was not a relevant part of our investigation.

Since we are only implementing three different programs in each of the different languages, the results will not cover all aspects of all the environments.

We only tested our programs on one of the available cameras, which means that the results could possibly differ on another significantly different Axis Communications platform. We realize that if the tests are conducted on another platform outside of Axis Communications, the results might also differ significantly. At least the absolute numbers.

Important to note is that while we refer to the performance of Java and C# in this report, in reality we are actually testing specific implementations of their virtual machines. Other implementations may show different results. More information about the setup we have used can be found in section 3.1.

1.4 Contributions

The work we have carried out in this thesis contributes by providing realistic benchmarks of different languages running in embedded systems having similar performance as the device presented in section 3.1. This contribution consists of our own test suite with typical use case applications for the devices in question, as well as measurements of what we consider relevant performance factors. We also provide a discussion about the trade-off between performance and programmer productivity in order to be able do determine the feasibility of running higher level languages in resource-constrained devices. This contribution is achieved by using our performance results as well as a thorough investigation about language and runtime differences. Another contribution is that we further explore the question regarding the use of C# in embedded systems. This topic has received much attention in research for Java, but not so much for C#.

Other minor contributions found in this report are listed below.

- General details of how to get Mono and Java to run on the architectures used by Axis Communications.

- Ideas on how to make the Mono footprint smaller for Axis Communications.

1.5 Division of Work

Most of our work was done in parallel and the distribution was fair. Although, there were some key differences to what work we did. This section presents the workload distribution for the practical work as well as the report writing.

1.5.1 Workload Distribution

Daniel

- Downloaded, compiled and integrated all open source frameworks needed for the test programs to run.
- Wrote motion detection program for C# and Java.
- Wrote snapshot program for C with the help of `libcurl`.
- Wrote both DOM and SAX parser programs for C#, C and Java.
- Ran the result parsing script in Python to format the results from the tests.

Philip

- Successfully compiled and tweaked the 17 dependencies needed for `libgdiplus` needed for Mono.
- Modified makefiles in order to get Mono to compile correctly.
- Wrote motion detection program for C.
- Wrote snapshot program for C, C# and Java.
- Monitored the tests and collected them for formatting.

Both

- General investigation about Mono and how to compile it for the cameras.
- Investigation of JVM alternatives suitable for the camera architectures.
- Gathered information about the cameras.
- Interviewed Axis Communications employees.

1.5.2 Report Writing Distribution

Daniel

- Most of the sections in chapter 2 from section 2.3 to the end of the chapter.
- Small parts of chapter 3.
- Everything in Results, including scripts to generate the figures.
- Conclusion and Recommendation in Summary.

Philip

- Previous Work in Introduction.
- Contributions in Introduction.
- From start of chapter 2 to section 2.3.
- Large parts of chapter 3.
- Future Work in Summary.

Both

- The Abstract.
- The Acknowledgements.
- Background in Introduction.
- Limitations in Introduction.
- Corrections and reformulations throughout the whole report.
- The Discussion chapter was equally contributed to.
- The Bibliography.

Chapter 2

Programming Languages and Execution Environments

This chapter will present several concepts that are important for the work described in this report. It will also include the general concepts of programming languages as well as differences between them in terms of performance, ease of use and maintainability.

2.1 What Defines an Embedded System?

It can be hard to pinpoint exactly where the line between an embedded system and a personal computer is drawn. It can depend on how the performance is in terms of CPU power, flash memory, RAM, operating system, or simply the fact that the physical chip is used for one or two use cases. Sometimes the device has an operating system, like the Axis Communications cameras, and sometimes, they do not.

Even though there can be major differences between different embedded systems, Burns and Wellings identify an embedded system with these 8 basic characteristics [14]:

- Largeness and complexity
- Manipulation of real numbers
- Extreme reliability and safety
- Concurrent control of separate system components
- Real-time control
- Interaction with hardware interfaces
- Efficient implementation

As the technology moves forward, this might change. A system that has a running embedded platform may be controlled by a more robust complete computer in the future. The homepage of the Electrical and Computer Science Department of NC State University states that [15]:

> ”An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls. Unlike a general-purpose computer, such as a personal computer, an embedded system performs pre-defined tasks, usually with very specific requirements.”.

While the definitions vary, common for most definitions is that embedded systems usually run on hardware that is resource-constrained, usually due to lack of performance, but also due to unavailable hardware components, as opposed to general purpose computers. When we talk about embedded systems in this report we therefore mean systems that have a limited amount of resources, mainly in terms of performance. In our work, it is the cameras at Axis Communications. Since the cameras are powered via the ethernet cable, they do not need to be energy conservative in the same way as a device with a battery is, for example a mobile phone. They are limited in flash memory and RAM. For full specifications on the hardware we conducted our tests on, see section 3.1.

In many embedded systems, there are different levels of real-time requirements. What those different levels are defined as can be read about in section 2.6.

Many people today might not think that there are that many different types of embedded systems, but as a matter of fact, embedded devices are all around us - 99% of all processors are made for the embedded market [16]. Examples of embedded systems on different levels are: ATMs, mobile phones, computers in cars and airplanes, weather sensors and cameras.

2.2 Programming Languages

There are many different programming languages that can be used to produce executable code for a computer. Different languages are better suited for some task than others, and vice versa. In the context of resource-constrained platforms, the industry tend to choose high performance languages that run without a virtual machine, such as C or C++ [3]. In high end computers there is instead a preference to use higher level languages which normally is considered to be much easier to handle in terms of maintainability and ease of use, but at the cost of using more resources [7].

2.2.1 Performance Metrics

The performance of a computer program can be defined by multiple factors. Some common and important performance factors, especially in the context of embedded platforms, are listed below [7].

- The execution time in seconds from the start to end of the program.
- The memory usage in bytes during execution of the program.
- The processor usage in percent during execution of the program.

- The size of the program in bytes when stored on secondary memory.

2.2.2 Ease of Use

Modern high-level languages provide more language constructs and more abstract libraries than the ones found in low level languages such as C. This is done in order to allow the programmer to accomplish many common tasks with less effort, and thus improving the speed of the development cycle.

Ease of use can be seen as a multitude of factors. However, measuring these factors is not always trivial due to the fact that there is not always any concrete way to measure them. In the case with the physical performance one can often measure aspects using tools (RAM usage, execution time, etc.). For the more abstract ease of use measurements one has to define what is considered ease of use in a particular case and how to quantize it. Some possible measurements for ease of use are listed below [7][17].

- The development time required by a programmer to complete the task.
- Availability of libraries used to implement the program in a specific language.
- The opinion of a programmer having written the same program in multiple languages.

2.2.3 Maintainability

The maintainability refers to the difficulty of making changes to existing source code and how easy it is for one programmer to understand another programmer's written code. As an example, consider the extreme example of a program written in assembly and another program completing the same task in a modern high-level language. It is much harder for a programmer to understand and modify the low level program as opposed to the high-level language which has many abstractions in place. One line of code in a high-level language could correspond to hundreds, if not thousands, of instructions in assembly.

Measuring maintainability, just like measuring ease of use, is not trivial and requires measurements to be defined and quantized. Some possible measurements for maintainability are listed below [7].

- The lines of code needed to construct a program performing a certain task.
- The design decisions of the completed program.
- The comment density needed for a program to be understandable.

2.3 Execution Platform

As mentioned before, programs can be run in different environments. In this section, we will discuss the difference between running a program natively - to compile and run a program for a certain CPU architecture, versus running a program in a virtual machine.

2.3.1 Native Code

C and C++ are translated to native code at compile time. One can think of the native code as instructions in a certain language the CPU must be able to understand. If one wants to run the program on different systems, different compilers that produce different executable files need to be used. An advantage to this approach in terms of performance is that no code has to be translated during runtime. All code is already translated for the CPU to execute.

2.3.2 Virtual Machine

A virtual machine is a program which can interpret other programs and execute them. This allows the virtual machine to provide a high-level abstraction to the programmer to use both for escaping low-level programming problems as well as bring portability. A compiler takes in source code and produces a file with intermediate code that the virtual machine can read and execute as seen in Figure 2.1. If the virtual machine interprets a language that is independent of operating system and CPU architecture, the programmer only has to produce one variant of the executable code. One single Java program can run on a Macintosh, a Windows or a Linux machine given that there is an installed Java runtime system for the machine.

Since a virtual machine is an enclosed system, it can enforce limits. For example, the JVM does not allow any memory manipulation and that is one of the reasons it is harder to create destructive applications in Java than it is in C. It can also be seen as a disadvantage because it puts shackles on the programmer if he or she wants to use a feature which is limited.

More layers and more parts in a system can easily mean lower performance. This is an area which is always of interest when it comes to optimization.

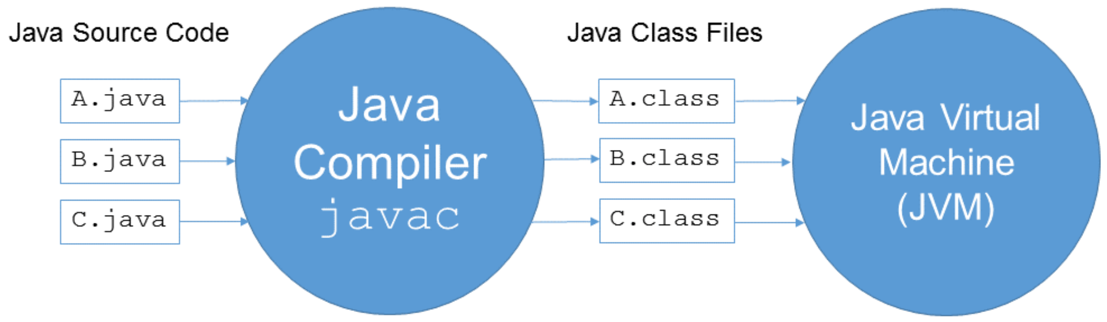

Figure 2.1: The steps needed for Java source code to be able to be executed is shown in the figure. The left circle is the Java compiler which produces class files needed for the Java virtual machine in order to run the program.

JIT and AOT

Code ultimately runs on the CPU. Somehow, somewhere, it has to be translated to native machine instructions. When interpreting code in an intermediate state such as Java code compiled into bytecode and C# code compiled into Common Intermediate Language (CIL), the translation to native code can be somewhat slow. In order to speed up this process, techniques such as JIT (Just-in-time compilation) and AOT (Ahead-of-time compilation) have emerged. They move the translation down to native code away from the time critical moments. Even though we would need an entire chapter to discuss differences, a basic distinction can be summarized like this:

- Ahead-of-time: Before application launch. If translating as much as possible of the code to native code at application launch, the system does not need to spend as much time on that during runtime. Even though this approach makes the application quicker to run, this takes up a lot of memory. In embedded systems, saving memory during runtime is often of great interest.

- Just-in-time: A JIT engine tries to translate smaller blocks of code in real-time before they are used. This approach is slower but uses less memory. Compared to AOT, because of the fact that the JIT engine is run during runtime, it has knowledge of the current state and can therefore do other optimizations the AOT is not capable of doing. All Java classes are not guaranteed to be loaded at the start of an execution of a program and therefore, a JIT approach fits better for Java than AOT [18].

Garbage Collection

If there is room for keeping track of allocated space in a runtime environment or specifically in a virtual machine, it opens up the opportunity for managing the memory. Instead of explicitly having to free allocated memory, a *GC (garbage collector)* can clean up afterwards. Even though memory leaks are possible with some obscure techniques [19] [20], a GC continuously, during runtime, deallocates memory which is not accessible. For a single garbage collector, we can have multiple garbage collection policies. The different collectors and policies are designed for different purposes. Some of them are for resource-constrained devices and others are for high performance in concurrent environments. For example, some policies might clean memory in bursts and then sleep for a while, and other might run in parallel [21].

Intermediate Code

As previously mentioned, a virtual machine reads code which is in an intermediate state, which is the output from the Java and C# compiler. Below are two examples of intermediate code formats.

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2
       3: ldc           #3
       5: invokevirtual #4
       8: return
}
```

Listing 1: In Java, the code is compiled to Java bytecode. A class file contains Java bytecode, along with other things, and is run in the JVM. The instruction set was designed to be compact [22]. This code is a normal ”Hello World” program compiled with `javac` and then disassembled with `javap`.

```
.assembly Hello {}
.assembly extern mscorlib {}
.method static void Main()
{
    .entrypoint
    .maxstack 1
    ldstr "Hello, world!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Listing 2: In C#, the code is compiled to Common Intermediate Language. The Common Language Runtime (CLR) executes this. This code is a normal ”Hello World” program.

2.3.3 Virtual Machines Versus Native Code

One important aspect to consider that can differ between programming languages is that some of them run directly on the machine hardware, i.e. the source code gets compiled to native code for a target platform. C and C++ are examples of this. Other languages require that the code is compiled into bytecode and run on a virtual machine executing on the target platform, for example Java and C#.

Running native code is in general faster than running code in a virtual machine. However, virtual machines are often capable of doing optimizations that compiled native code can not do. They are also capable of techniques such as Just-in-Time (JIT) compilation and Ahead-of-Time (AOT) compilation which produce native code based on the bytecode passed as input to the virtual machine. While virtual machines have many benefits, running them is not free in terms of resources. They often consume a lot of RAM and interpreting the bytecode is much more expensive than running natively. Due to this, it is often preferable to use native code in resource-constrained environments.

2.4 Expressive Power of Programming Languages

The expressive power of programming languages can be seen as the ability for the programmer to express his or her ideas in language constructs. There is no formal framework for comparing the relative expressiveness of programming languages [23], but it is still a topic often appearing in literature. In this section some key differences between the expressive power of the investigated languages will be presented in order to motivate why there is a significant difference in expressiveness between low- and high-level languages.

2.4.1 Object Orientation

Both C# and Java are treated as big OOP (object oriented programming) languages. They have great support for classes, inheritence, polymorphism, abstraction, encapsulation, and more. The closest C comes to OOP is through structs, which, when compared to classes are very lacking in functionality.

2.4.2 Generics

When creating classes that should have a relationship with another class, for example if a class should contain a collection of another class, generics are a perfect fit for the problem. Even though, their implementation and usage is somewhat different, both Java and C# has support for it. Constraints and requirements can be specified for parameters, return values and member variables.

2.4.3 Exception Handling

A good mechanism for handling possible errors and exceptions when writing code is **exception handling**. Being able to throw exceptions and force the calling method to catch them encourages a good usage of a method. Exception handling is rich and functional in both C# and Java, however in C, there is no support for it.

2.4.4 Pointers

In Java, there are references/pointers and there are values, but the programmer can not manipulate where the pointers should point, only follow the reference. The behavior is the same C# unless the `unsafe` keyword is used in a method declaration. If it is used, the programmer is free to manipulate pointers as if they were programming in C.

2.4.5 Operator Overloading

If one wants to add an object with another, **operator overloading** can be useful. Below is an example of how it can be used in C#. `ComplexNumber` is a class that should represent a complex number ($c = a + bi$).

```
class ComplexNumbers {
    public static void Main(string[] args){
        /**
        * The constructor of ComplexNumber takes
        * the real part as the first argument
        * and the imaginary part as the second argument.
        */
        ComplexNumber p = new ComplexNumber(4, 5);
        ComplexNumber q = new ComplexNumber(-1, 10);
        ComplexNumber c = p + q;
    }
}
```

Listing 3: In this example, a proposed use case of operator overloading is shown in C#.
`ComplexNumber` is a class that should represent a complex number ($c = a + bi$).

The programmer would not have to extract the real part and the imaginary part and then create a new object from it.

Unfortunately, if the programmer would try something like this in Java or C, he or she would encounter a compilation error. The behavior can simply not be defined in any of those two languages. However, in C#, the programmer can define what should happen when adding, subtracting, multiplying or applying other operators on objects of a class.

2.4.6 Unsigned Data Types

In C# and C, the programmer has the ability to choose whether he or she wants to have their primitive numerical data types to be unsigned or not. This is not possible in Java.

2.4.7 Reflection

Reflection, which is the act of a program reading and modifying its own structure and properties, is available in Java and in C#, but not in C.

```java
public class Reflection {
    public static void main(String[] args){

        MyClass obj = new MyClass();
        Method[] method = obj.getClass().getMethods();

        for(Method method : methods){
            method.invoke(obj, null);
        }

    }
}
```

Listing 4: Reflection, the act of a program reading its own structure. The code in in Java. This code retrieves all methods in a certain class, `MyClass`, and invokes all of its methods.

2.4.8 Typing

All three of C, C# and Java are statically typed [24][25]. Although, with the keyword `var` in C#, it is possible to let the type of a variable be inferred by the compiler, even if it is still statically typed. Java 8 also features some limited type inference [26]. An example of type inference in C# is shown in Listing 5 and in Listing 6.

```csharp
// This statement:
var s = "Hello World";

// Is equivalent to this statement:
string s = "Hello World";
```

Listing 5: Basic functionality of the keyword `var` in C# is shown in the listing above.

```
// Instead of this:
ALongNameForAClassWhichIsAnnoyingToType x
    = new ALongNameForAClassWhichIsAnnoyingToType();

// We can type this:
var x = new ALongNameForAClassWhichIsAnnoyingToType();
```

Listing 6: In the example above, we show how the keyword `var` in C# can be used to free the programmer from writing too long class names in variable declarations.

2.4.9 Lambda Expressions

Lambda expressions are a form of simplifying expressions so the programmer does not have to write out a complete method. The idea comes from functional languages. It can be useful when using methods as a parameter to another method as well. Lambda expressions are available in both C# and Java (from version 8). See example below taken from a post by a user on `StackOverflow.com` [27]:

```
// We can write this:
string person =
    people.Find(person => person.Contains("Joe"));


// Instead of this:
public string FindPerson(string nameContains,
            List<string> persons)
{
    foreach (string person in persons)
        if (person.Contains(nameContains))
            return person;
    return null;
}
```

Listing 7: In the example above, we can see how we can use lambda expressions to write smaller programs which do the same thing.

2.5 Other Notable Differences Between C, C# and Java

Aside from the power of expressiveness in the different languages, there are also other important aspects that can influence the productivity of the programmer. Examples of this could be features available in virtual machines for Java and C#. In this section, some differences will be presented in order to further motivate the use of higher level languages when possible.

2.5.1 Garbage Collection

Since both C# and Java run inside a virtual machine, there are garbage collectors to clean up after the programmer. In C, you have to explicitly `free` the objects you want to remove from the heap area in the memory. In both C# and Java, the garbage collector can be called explicitly, but it is often run in the background without anyone noticing when it cleans up.

2.5.2 Calling C Code

If the system partly consists of C code that needs to run and the code that is being developed is not in C, one still might want to call the old code. There are ways of doing this in C# and Java, depending on platform. In Java, JNI (Java Native Interface) is used and in C#, one line of code has to be written that refers to the compiled C code in an `.so` file.

2.5.3 Cross-Compilation

As previously mentioned, when a program executes, a processing unit reads and executes instructions. Different CPUs read different sets of instructions. These instruction sets differ between different CPU architectures. Close to all desktop computers today run on the x86 CPU architecture so when a program in C is compiled to what we refer to as native code, which is CPU instructions, we may only run it on the architecture we compiled it for [28][29]. Seeing as many embedded systems do not share the same architecture as the build computer's architecture, a program cannot simply run natively on both types of architectures. Therefore, we introduce something called cross-compilation. It means that we compile for a certain architecture, a target machine, on a build machine. In our case that meant building ARM programs on a x86 machine. The hard part about the process is that when a program is compiled, it cannot be run and verified that it works on the build machine. It has to be transferred to the target machine. The command `file` [30] for scanning the file for common flags that tells us what architecture it is targeting, although the information may not always be sufficient. There are tools out there to help with the process of cross-compiling, such as **Scratchbox** [31] which is a toolkit project. The process is not trivial and it is a science in itself.

This is where C# and Java comes in as a savior. If a C#/Java program works and is compilable on a build computer, the only things that needs to be done it simply copying it over to the target machine without hassle.

We encountered some problems when building and compiling applications within the ecosystem of Axis Communications. The build tool `make` together with `AutoGen` is used and without any prior knowledge of how it works, it can be hard to understand how it works. In order to identify which parts that should have been altered, we had to dwell through thousand lines of auto-generated make files. What we did should be considered a workaround and to correctly integrate the compilation process into the system, a knowledgeable person should have done it. It was a tedious process and it really taught us that avoiding cross-compiling can spare many hours of frustration.

2.6 Real-Time Requirements

Different systems have different requirements. Some systems, such as collision avoidance in a car or damage prevention in a machine in a factory often have strict requirements when it comes to time delay. If they do not fulfill their requirements, severe consequences, such as someone being injured, may occur. In other systems, such as a camera that is scheduled to turn off during weekends, the time requirements are not as strict since there are no severe consequences if the camera is turned off a second later than scheduled. At Axis Communications, close to all use cases fall under the category of soft real-time requirements.

2.6.1 Hard Real-Time

When talking about **hard real-time**, if a deadline is not met, we have experienced a system failure. Emergency and automatic control systems are often considered as hard real-time. These exist in airplanes, nuclear power plants and in pacemakers. Their purpose is to be extremely quick to either shut down, trigger and emergency calls or procedures, or simply just do their designed task within fractions of seconds.

Truly, hard real-time systems are uncommon. This is advantageous for the work in this thesis, as it does not apply for hard real-time systems, but it is still of great relevance due to the fact that most systems are not hard real-time simply because we have more wifi light bulbs, kitchen appliances and cameras than nuclear power plants.

2.6.2 Firm Real-Time

Firm real-time systems allow some missed deadlines every now and then. If a few of the messages happen to be delayed, and these messages are not crucial to the system functionality but still of importance, we call it a firm real-time system. A surveillance system that is supposed to take a snapshot of a burglar when a store is broken is one example. If the system misses to retrieve the frame where a face can be identified, it does not crash, but it provides no useful information.

2.6.3 Soft Real-Time

When a system still works and the information is useful even after a deadline, we have a **soft real-time** system. In a weather station with sensors for wind speed, or if you have automatic awnings that should extend when the sun comes out, you can afford to miss a deadline. Even though the outcome is suboptimal, value can still be gained from the triggered process.

Chapter 3
Method

This chapter will describe the process we used to execute the investigation together with what equipment we used. The work was performed in different phases. The first phase revolved around investigating the equipment and getting high-level languages to run on the platforms. The second phase was to implement test programs in order to evaluate the performance of these languages. The third phase was to define and carry out measurements on these programs. Finally, the last phase was to evaluate these measurements and reach a conclusion.

3.1 Setup

The tests were conducted on as up-to-date hardware and software as possible. This section will present the tools and resources we used as we carried out the experiments, and in-depth details of what versions and specifications they have.

3.1.1 Hardware

For our tests we used an, to that date, unreleased camera from Axis Communications from the M30 series. It is a fixed mini dome designed for retail stores, hotels, schools, banks and offices with tight budgets for video surveillance [32]. It has support for HDMI for live streaming to an HDTV monitor, a resolution of up to 1920x1080, 256 MB of RAM and 128 MB of flash memory and SD card slot.[33]. It runs an ARM Cortex-A9 CPU implemented by Ambarella from their S2L family designed for image processing. Their system on chip brings advanced High Dynamic Range (HDR) processing, intelligent video content analytics, and wide angle viewing into mainstream professional and consumer IP cameras [34].

The ARM Cortex-A9 CPU is, as ARM states, a popular general purpose choice for low-power or thermally constrained, cost-sensitive devices. It is also popular in smartphones, digital TV and IoT (Internet of Things) devices. The CPU is a single core processor and implements the ARMv7-A architecture [35].

The camera we used was using a custom version of Linux, tailored to Axis Communications' needs.

The build computers we used were running Windows 7 Enterprise with a virtual machine running Linux Debian 4.6.3-14.

3.1.2 Software Tools

The following software was used during the course of our tests.

GCC

GCC was used both to compile benchmark C programs as well as compile Mono. The one used is an internally Axis Communications modified version. It is based on gcc version 4.7.2 20120820 (prerelease) [gcc-4_7-branch revision 190527].

Mono

The configuration and version of Mono can be seen below in the printout of `./mono-sgen -version`.

```
Mono JIT compiler version 4.2.2 ((no/996df3c Mon Feb 1 11:02:27 CET 2016) Copyright (C) 2002-2014 Novell, Inc, Xamarin Inc and Contributors. www.mono-project.com

        TLS:  __thread
        SIGSEGV:  normal
        Notifications:  epoll
        Architecture:  armel,vfp+fallback
        Disabled:  attach,com,debug,full_messages,logging,profiler
        Misc:  smallconfig softdebug
        LLVM:  supported, not enabled.
        GC:  sgen
```

JVM

The configuration and version of the JVM can be seen below in the printout of `./java -version`.

```
-XX:InitialHeapSize=5242880 -XX:MaxHeapSize=100663296 -
XX:+PrintCommandLineFlags
java version "1.8.0_65"
Java(TM) SE Embedded Runtime Environment (build 1.8.0_65-b17, profile compact1,
headless)
Java HotSpot(TM) Embedded Client VM (build 25.65-b01, mixed mode)
```

The Java runtime used two different garbage collection algorithms, `MarkSweepCompact` and `Copy`.

3.2 Language Platforms

Different programs run in different runtime environments. C# and Java programs are compiled to an intermediate state between machine code and source code where the compiled files can be read by their respective virtual machines. C on the other hand is compiled to machine code.

3.2.1 C

At compile time, C programs are built for a certain architecture. Axis Communications' own modified version of GCC[36] was used to compile these programs. We do not believe that the changes have any effect on our work.

3.2.2 C#

C# programs were run in Mono, an open source implementation of Microsoft's .NET Framework [37][38]. The Mono framework was compiled for the 32 bit ARM CPU architecture used in our cameras with Axis Communications' own modified version of GCC.

Only a subset of classes were available for use within Mono from the basic install. Along with Mono exists a standalone library module called *libgdiplus* which complements the missing parts of the basic Mono install [39]. This module is an implementation of Microsoft's GDI+ API for non-Windows based operating systems. It is responsible for providing some graphic routines used by the .NET framework. While the target platform in our study does not have any graphical output, this library is still required in order to use some of the classes included with the standard library, such as the `Bitmap` class. The libgdiplus library is thus required in order to have a full Mono install with high compatibility.

3.2.3 Java

Java programs were run on a JVM provided by Oracle that was compiled for a 32 bit ARM CPU architecture with floating point numbers calculated in software instead of hardware [40]. The reason for this is simply because Oracle did not provide a version of JVM with hardware calculated floating point numbers for our CPU.

3.3 Investigation and Installation

To carry out the investigation, a decision was made to study the high-level languages C# and Java. Both of these languages have a large user base and many competent programmers on the market, thus making the languages attractive options for our study. They both offer many interesting features of modern object oriented languages. They also both run in a virtual machine capable of technologies such as AOT or JIT, which is very important for good performance.

Since vital aspects such as the CPU architecture differed between the available platforms at Axis Communications, the support for the runtimes of the high-level languages also differed. When the target platforms had been chosen we had to set up the environments needed in order to run these languages on them. A significant workload was put into researching, building and testing different setups in order to produce a promising environment for the language runtime systems.

3.3.1 Setting Up the C# Environment

The .NET framework and C# have mainly been of interest for platforms running Microsoft Windows. At the time of this writing there is no full official implementation of this framework available for platforms running Linux, as the target platforms in our study do. Thus we had to resort to running a non-official open source implementation of the .NET framework which aims for full compatibility. There are some implementations that could be considered interesting alternatives such as the *.NET Compact Framework* or the *.NET Micro Framework* [41] [42]. These alternatives does not offer a complete implementation or lack important performance features such as AOT or JIT and was considered to be insufficient for our purposes.

The most interesting and cross platform compatible full implementation of the .NET framework that we found was the Mono project. While it at the time still lacks some features available in the .NET specification, it has become mature in its development and can be considered to be in a commercially usable state. The project also aims for complete coverage of the .NET specification with a few exceptions. These exceptions are the *Windows Presentation Foundation*, *Windows Workflow Foundation* and *Windows Communication Foundation* [43]. All of which are non-important for the target platforms under investigation because there are no needs for graphical user interfaces or communication with IIS servers.

In order to use Mono, we had to compile the complete project from source. All platform independent files were compiled using an x86 computer, but the platform dependent files had to be cross compiled using the toolchain available for the Axis Communications cameras. When compiling from source, many different compile time flags were examined in order to optimize the binary for the target platforms. Many of the flags disable unnecessary features in the context of our study and reduces the size of the binaries and libraries. For complete compile instructions for our installation of Mono together with libgdiplus, see our guide in our GitHub repository [44].

The difficulty with compiling libgdiplus was that it had many library dependencies not available for our target platforms. Most of these dependencies were related to *Xlib*, a library used by the *X Window System* found on most Linux platforms with graphical output [45]. As a result, in order to get libgdiplus up and running we had to compile Xlib, its dependencies, and a few other libraries using the toolchains available for our target platforms.

3.3.2 Setting Up the Java Environment

Getting the Java runtimes up and running on the target platforms did not pose much of a problem as Oracle provides pre-compiled binaries of Java for ARM processors. Running Java was thus simply a matter of downloading the binary and accompanying libraries and transfer them to the target platform.

3.4 The Test Programs

We constructed suitable test programs in order to evaluate the high-level languages on the target platforms. These programs were chosen based on what could be typical use-cases for the software developers at Axis Communications. Since the languages differ and we take more factors into account than just the exeuction performance, the test programs for each language are written in a completely independent way. This means that each program is written as a programmer in that language would have written it, thus the internal algorithms can differ between languages, but the problem solved by each program is always the same. Implementation details of the algorithms are presented in section 3.6. The test programs we chose are presented in the following subsections. Parts of the implementation, which does not expose anything Axis Communications does not wish to present to the public, can be found at GitHub [44]. For all examples, except for the motion detection program implemented in C, the missing parts are obvious and can be implemented trivially. The motion detection program for C is completely omitted because of its strong dependency to proprietary software.

3.4.1 XML Parser

While XML parsing is not something Axis Communications specializes in, it is still a task that often needs to be done in a variety of programs and not something that Axis Communications wants to spend too much time on solving. Instead, they want to focus on the problems that actually increase the amount of customers. XML parsing can also be a rather heavy task depending on the size of documents in need of parsing, thus making it a suitable test program for our purposes.

Two different XML parsers were made, one *Document Object Model* (DOM) parser in which all of the document contents gets stored in memory as a tree representation which the programmer can modify as wished. The other one was a *Simple API for XML* (SAX) parser which traverses the document once and performs actions as they are needed.

The task for both of these parsers was to find the deepest node of the document. Having a simple task like this allows us to focus the computation on the actual parsing and not the use of the parsed data.

The XML file used for the DOM tests was 13.8 megabytes in size and the file used for the SAX tests was 23.9 megabytes. We chose these sizes in order for the tests to run within a suitable time frame and use a reasonable amount of memory. In real situations at Axis Communications, these XML files are usually at around 0.5-3 megabytes.

3.4.2 Snapshot

The task of the snapshot program was to take a snapshot image with dimensions 1920x1080 pixels from the camera mounted on the target platform and then send it to a specified FTP server. The snapshot was grabbed using the camera HTTP API found in Axis Communications camera SDK. We chose to repeat this task ten times for our tests since it is not very resource- or time-consuming if only done once. This had to be done in order to have measurable results. It also enabled us to reduce the noise in the results which comes from start-up time of the virtual machine and loading of classes.

Axis Communications often need to use the API to communicate with the camera hardware in order to perform operations on images and video streams. This test program acts as a small use-case for this where the taken image is transferred to another computer via FTP. This could for example be used to send an image to some user's computer when motion has been detected in a video stream from the camera in question.

3.4.3 Motion Detection

A very important feature in modern security cameras is the motion detection. If something moves in a stream of images, the camera should be able to tell if this happens, where and when it happens. Analyzing a stream of images is not trivial work, especially not for a resource-constrained embedded device. A lot of processing power is needed in order to constantly process new images arriving from the camera hardware. The importance for Axis Communications and the high performance requirements thus make motion detection a very suitable test case in our study.

Since computer vision is a whole field of study by itself, we have not designed any motion detection algorithms by ourselves. We have chosen to use different libraries which for each language natively implements a motion detection algorithm. This of course means that these algorithms can differ a lot in quality and execution. But since we are looking at the results from a more abstract view point, and also taking more factors than just performance into account, the results should still prove to be interesting for the question ”how well does motion detection perform on the resource-constrained device?” for a given language platform.

In practice, the motion detection algorithm would most often run in real-time in order to detect events as they happen. However, since we wanted to test the same input for the different programs, we found it to be better to use a local file instead. This has the benefit of making our tests more consistent. Since there is no timing involved in this case, we also get the benefit of looking at the processing time for all frames without delay.

The file we used in the tests is a file containing an MJPEG stream which was grabbed from an Axis Communications camera through HTTP by using the program `wget`. The video data is around 10 megabytes and contains a video with a resolution of 320x240 pixels. The video includes long sequences of no movements, every now and then movements as well as high frequent motion.

3.5 Test Suite

In order to produce valid test results we had to define the test suite that was to be used in all tests for all different languages. All tests were constructed in such a way that they ran for several seconds, this avoids the problem with lack of accuracy for small programs, and makes the startup time less significant. In order to further improve the accuracy, all tests were run five times each and the mean value of these are the ones presented in this report.

We used two tools in order to do the actual measurements on the execution time, CPU usage and memory usage. One of these was the common Linux program `time` which calculates the execution time of the program given as an argument to it. The second tool was an Axis Communications internally developed python program called `hogclient` which simply runs several Linux commands with a sample rate of about one second. The information grabbed by `hogclient` for our study were given by the program `iostat` and the information in `/proc/meminfo`. Information about the current CPU load comes from `iostat` and the details about memory usage from `/proc/meminfo`. Since we wanted the data to be formatted in an easy way so we could visualize it in diagrams, we took the output from `hogclient` and sent it through our toolchain of python scripts which normalized the results, formatted it in the way we wanted and produced the output diagrams.

3.6 Implementation of Test Programs

This section will provide details about the implementation of each of the test programs for all languages. These details might need to be taken into account when comparisons are made between languages, since all implementations are different due to language and library differences. The following subsections will for each program present key differences in the programs that could cause differences in measured data.

3.6.1 XML Parser

Different libraries were used in order to implement the XML parsing functionality for the different languages. As stated in Section 3.4.1, two different implementations of the parsing were made, one using SAX and one using DOM. The Java and C# implementations managed to complete this task without the use of external libraries, but the C implementations had to use two open source external libraries, one for each type of parsing.

C

The implementation of the DOM parser for C uses a library named *ezxml*. This library provides a very simple API for parsing DOM trees, and while it is sufficient for our simple test program, it has its limitations. For one, ezXML is not a validating parser, meaning that it will only check for syntax errors which makes it a bit faster at the cost of some accuracy. Secondly, it requires the input XML file to be loaded in full, and will thus not allow it to be loaded in smaller chunks at the time. This has an impact on how memory usage will look during the loading of the file.

The C implementation of the SAX parser uses a library called *expat*. The test document is read in five megabyte chunks in order to have similar memory usage as the implementation for the other languages. Like ezXML, expat is also a non-validating parser and will thus be a bit quicker than validating parsers.

C#

Both of the SAX and DOM implementations for C# were done using only standard libraries. The DOM variant was implemented using the `XmlDocument` class found in `System.Xml`. The SAX variant was done with the class `XmlReader` to iterate over the XML file, also found in `System.Xml`.

Java

Like the C# versions, the Java implementations were also done using only standard libraries. The classes used for the implementation were found in the packages `org.w3c.dom` and `org.xml.sax`. These libraries support the use of XML DTD files and are thus able to create validating parsers. However, our tests does not make use of any DTD, thus making the parsers non-validating just like in the other implementation of the programs.

3.6.2 Snapshot

The snapshot program was implemented using an HTTP API for the target platforms. As long as the test programs can handle HTTP with digest authentication, then the API will work for all of them. Connecting to HTTP, and FTP for the uploading phase, are handled a bit differently in the different languages. The most important differences between these implementations are outlined below.

C

The C implementation was the only implementation of the snapshot program that had to rely on a third party library in order to have reasonable HTTP and FTP functionality. We chose to use the open source library *libcurl* to handle both HTTP and FTP communication at a higher level. One simply provides the parameters needed for communication such as username, password and host.

C#

Implementing HTTP and FTP functionality in C# was easily done with the `WebClient` class from the standard library. There is not much to say implementation wise for this other than that it was done using the functions `DownloadFile` and `UploadFile` of WebClient to achieve all network communication needed by the test program.

Java

Similar to the C# implementation, the Java program was easily implemented using classes from the standard library. The classes used to achieve the network communication were

URLConnection and a derivative of it called HttpURLConnection. A difference from the C# implementation was that these classes forces the programmer to use the InputStream and OutputStream objects received from them in order to download or upload a file. This requires a custom buffer in order to be done efficiently, and is thus a bit more complex than C#'s WebClient with the functions DownloadFile and UploadFile. The custom buffer can also have a small impact on the performance in case it differs from the internal library implementations in the other languages.

3.6.3 Motion Detection

Implementation of effective motion detection algorithms is not trivial work. Writing such an algorithm by ourselves in order to have a more fair comparison between the different languages would have been out of scope for this thesis work. We therefore chose to only make use of different native computer vision libraries for each of the languages. These libraries can be very different in how they are implemented, and this section will aim to explain as much of the differences as possible in order to have a more fair analysis on their performance.

C

The C implementation of the motion detection test program was done using a computer vision library developed at Axis Communications called *vision*. This library makes use of the open source library *RAPP*, also written by Axis Communications. RAPP provides optimized low level image processing functionality, suitable for implementing computer vision algorithms [46].

The test program we wrote using the vision library went through every single image from the MJPEG stream and processed them in groups of 16, as that was what the library's analyzing function required. For each call to the analyzing function, the result of detected motions could was received in the form of a black image which marks the detected motion areas with white. If the image contains any white, then it means that a motion has been detected. We chose to not process this image in anyway as our purpose was only to test the performance of the motion detection algorithm, but not actually use the data. This is an important difference because the other computer vision libraries for the other languages does not provide the detected motion in this way.

C#

For the C# implementation of the motion detection test program, an open source library called *AForge.NET* [47] was used. It is developed in modules which makes it easy to plug in different components. The program continuously returns decoded bitmaps from a stream and then the programmer has to feed these images into a motion detection algorithm component. We chose the simplest of them all, the **TwoFramesDifferenceDetector** class. The motion detection method returns a value between 0-1 that describes how much difference there was from one frame to another, where 0 is none and 1 means that all pixels changed. This value can then be compared to a threshold the programmer herself chooses.

Java

For Java, we used an open source library called *Webcam Capture* [48]. To use the library, you simply provide a stream and an object of a class that implements the interface `WebcamMotionListener`. The method that the class enforces the programmer to implement, is then called when a motion is detected from the stream of images that it provided to the library. In order to customize and change threshold value, the code has to be modified and recompiled.

Chapter 4
Results

In this chapter, results from the test program executions and the environments are shown. For the test programs, this includes CPU usage, RAM usage, execution time and size on secondary memory. For the runtime environments, size on secondary memory of them and startup times are included.

4.1 Size of Runtime Environments

Since compiled C programs can be run natively without installing anything from a clean install on the camera, the additional needed size on secondary memory is 0 MB. C# and Java need their respective virtual machines. The amount of memory they used can be seen in Table 4.1. This is discussed in subsection 5.1.5.

	Additional size on flash memory from clean install
C	0 MB
C# (Mono)	105.5 MB
Java (JVM)	14-61.4 MB (depending on included classes)

Table 4.1: The figure shows the additional size on secondary memory needed for the runtime environments

4.2 CPU Usage

In Figure 4.1, Figure 4.2, Figure 4.3 and Figure 4.4, the CPU usage consumed while running the test programs is displayed. Since the CPU usage stays at around 15% during idle, the total amount displayed in each of the diagrams never goes above around 85%. Some of the figures show points below 0% usage which simply is other programs using and stop using the CPU at the same time. Since the measurements were started and stopped manually with the help of hogclient, both start points and end points do not start and end exactly when the program launches and terminates but a little bit before and a little bit after. This is especially apparent in Figure 4.1 where we can see that the three last data points for C# are at 0% CPU usage. We can note that in all cases but the snapshot program case, the usage more or less stays the top capacity for the whole duration of the program.

In Figure 4.1, Figure 4.2 and Figure 4.4, the CPU usage stays constantly at the maximum value (around 85%) for the duration of the execution.

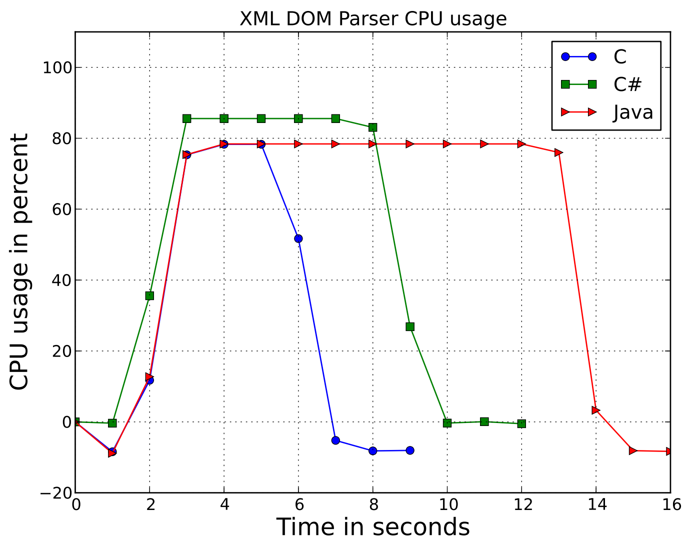

Figure 4.1: The figure shows the consumed amount of CPU in percent for C, C# and Java for the duration of the execution of the XML DOM parsing program.

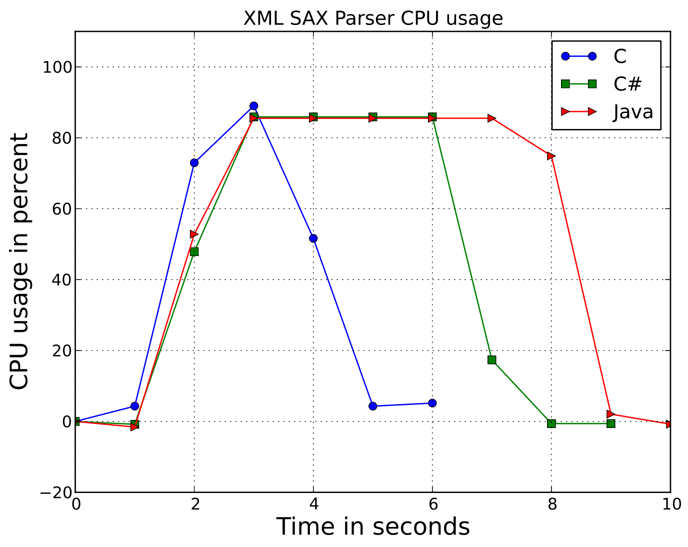

Figure 4.2: The figure shows the consumed amount of CPU in percent for C, C# and Java for the duration of the execution of the XML SAX parsing program.

In Figure 4.3 where we have a lot of networking calls where the program has to wait for response, we can see that CPU usage consumed only reaches a top for the C# program. For C and Java, the maximum points are at around 30% and 70% respectively.

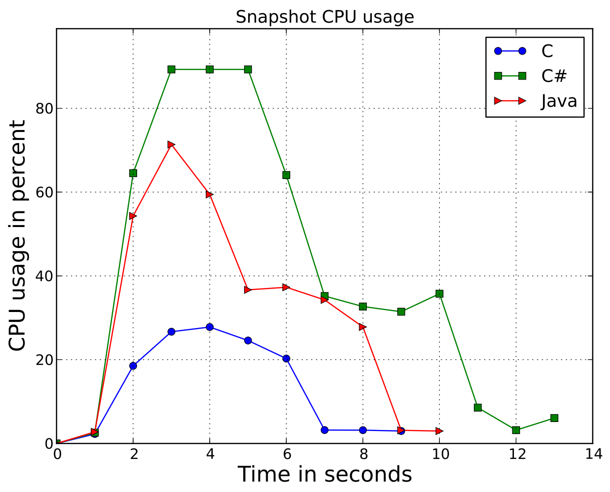

Figure 4.3: The figure shows the consumed amount of CPU in percent for C, C# and Java for the duration of the execution of the snapshot program.

Just as with the XML programs, in Figure 4.4, the CPU usage reaches a top where it stays until the program terminates. The result for Java stands out a lot. Discussions regarding possible reasons for this behavior is discussed in subsection 5.1.1.

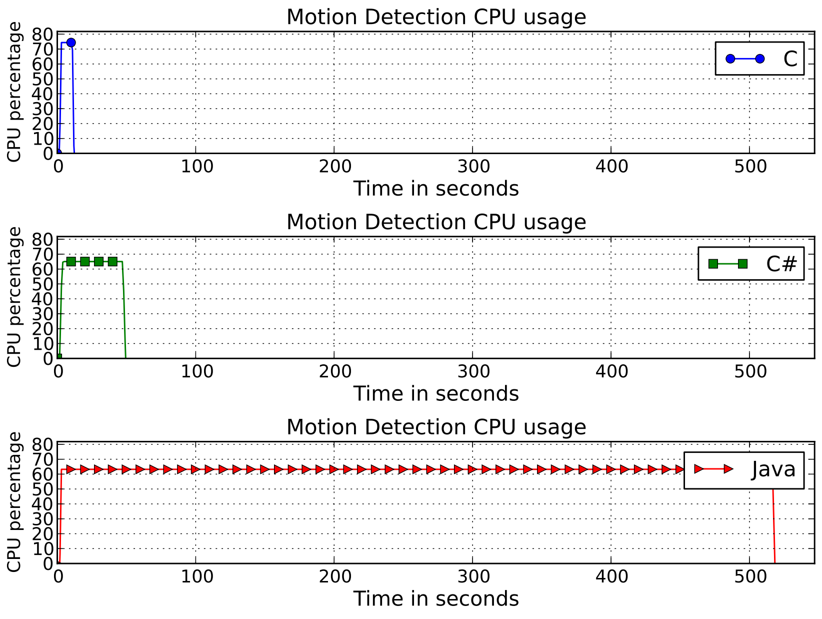

Figure 4.4: The figure shows the consumed amount of CPU in percent for C, C# and Java for the duration of the execution of the motion detection program.

4.3 RAM Usage

Similarly to the case with the CPU usage, hogclient was started and stopped manually and that is the reason for both start points and end points not starting exactly when the program launches and terminates but a little bit before and a little bit after.

In Figure 4.5, we can see that the memory usage over time scales somewhat linearly until the DOM tree is built and then the program terminates. The spike in Java memory usage is more than one would expect, and possible reasons for this spike are discussed in subsection 5.1.3.

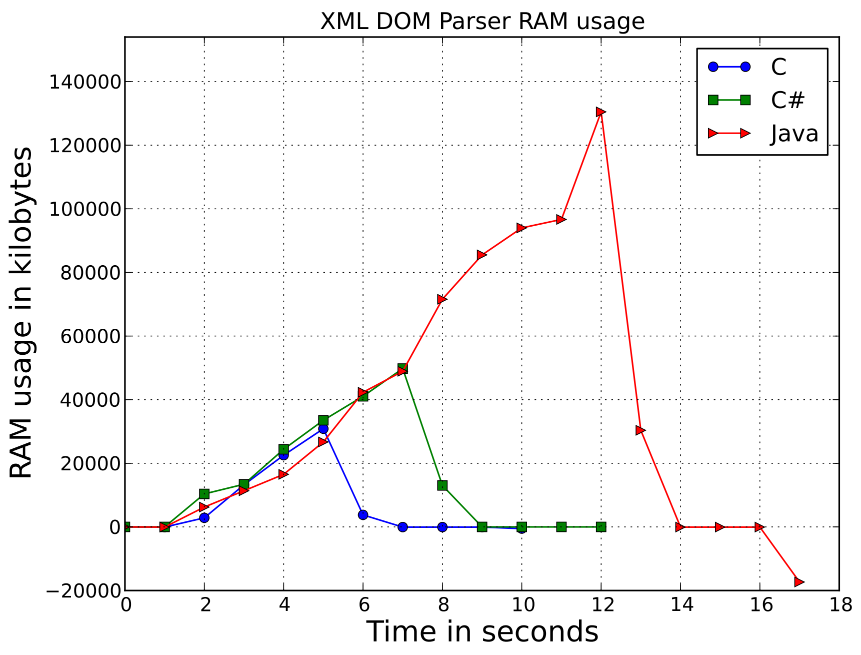

Figure 4.5: The figure shows the consumed amount of RAM in kilobytes for C, C# and Java for the duration of the execution of the XML DOM parsing program.

When traversing an XML tree with the SAX strategy, nothing is saved whilst traversing. Therefore we have a constant usage of memory.

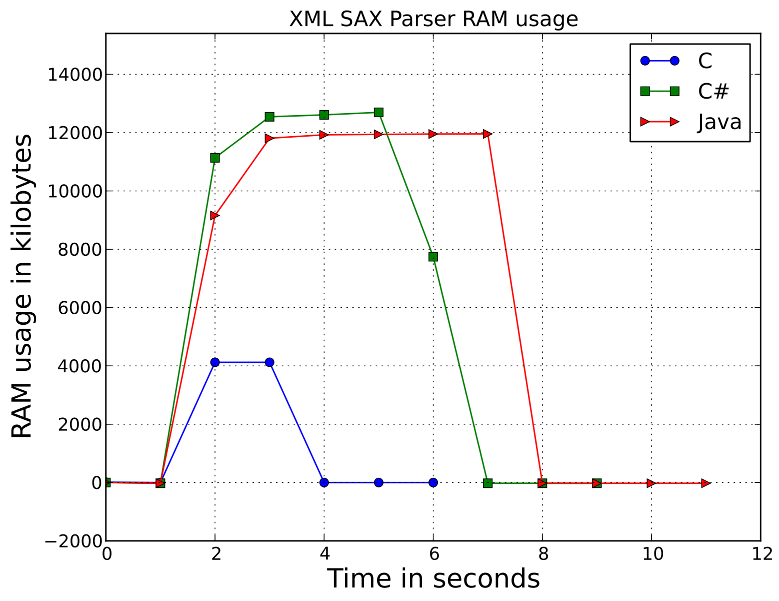

Figure 4.6: The figure shows the consumed amount of RAM in kilobytes for C, C# and Java for the duration of the execution of the XML SAX parsing program.

The amount of memory allocated increases over time until termination of the program for both C# and Java in Figure 4.7. This can be the garbage collector not intercepting with cleaning up the allocated memory. If run for an even longer time, the curve would probably reach a stable state. Since we did not run any profiling tools for the garbage collector, we can not say for certain whether this is the reason or not. For C, the amount of allocated memory stays constant. This is an effect of having control of explicitly freeing resources. As soon as an image is consumed in the program, it is freed.

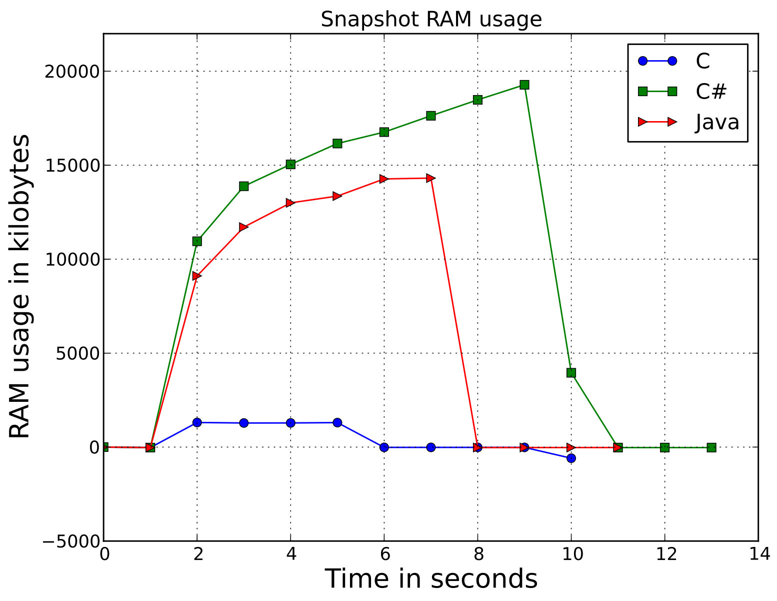

Figure 4.7: The figure shows the consumed amount of RAM in kilobytes for C, C# and Java for the duration of the execution of the snapshot program.

We can see in Figure 4.8 that the memory usage is constant for C and Java. In C, we have the same effect as in Figure 4.7 where the resources are freed quickly after they are used. For Java, the garbage collector is constantly busy freeing memory at the same pace as we allocate memory. For C#, the spiky tops are probably an effect of a lazier garbage collector than in Java. In this case, it does not free as aggressively as in the Java case. This can also been seen as the maximum values are higher here (almost twice as Java). The reason to why Java has such a long runtime is, as mentioned previously, discussed in subsection 5.1.1. The way the garbage collection acts can differ between implementations of garbage collectors, see subsection 5.1.3 for a longer discussion about this.

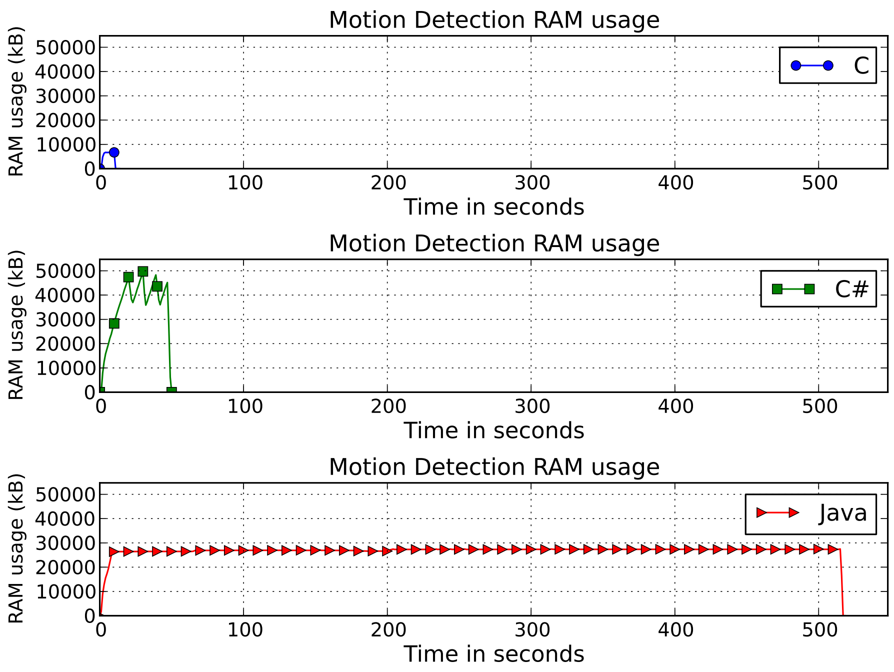

Figure 4.8: The figure shows the consumed amount of RAM in kilobytes for C, C# and Java for the duration of the execution of the motion detection program.

4.4 Execution Time

In general, C has the shortest execution time, followed by C# and then Java. Only in the case of the snapshot program, Java has a shorter execution time than C#, as can be seen in Table 4.2.

	C	C#	Java
XML DOM	3.91 s	6.67 s	11.29 s
XML SAX	2.33 s	4.75 s	6.51 s
Snapshot	4.36 s	8.78 s	6.27 s
Motion Detection	9.47 s	48.60 s	540.69 s

Table 4.2: The figure shows the execution times in seconds for all the different programs and platforms.

4.5 Size on Secondary Memory

For the XML DOM, XML SAX and snapshot programs, the C# and Java compiled files are significantly smaller than their C counterpart. Only in the case where we bring external libraries with the motion detection program, we have a larger compiled file for both C# and Java. The Java version is much larger than both the C and C# compiled files.

	C	C#	Java
XML DOM	39.0 kB	4.0 kB	3.8 kB
XML SAX	10.8 kB	3.5 kB	1.8 kB
Snapshot	18.0 kB	4.0 kB	3.1 kB
Motion Detection	263 kB	346 kB	2800 kB

Table 4.3: The figure shows the size on secondary memory in kilobytes for all the different programs and platforms, including all libraries needed.

4.6 Empty Program Execution Time

The time required to execute a program with an empty main method measured with `time` is shown in Table 4.4. For C, we have a result of less than 10 milliseconds because of no large runtime. For C# and Java, we have a runtime to start and therefore it takes some time. The time needed for the JVM compared to Mono is about 7 times as long.

	Time in milliseconds
C	< 10 ms
C# (Mono)	100 ms
Java (JVM)	730 ms

Table 4.4: The figure shows the total execution time for an empty program to start in the different runtime environments. This includes starting the virtual machine where applicable.

Chapter 5
Discussion

In all of the diagrams presented in chapter 4, it is easy to see that C outperforms both C# and Java in all of our test cases. Does that mean that we can conclude that we should throw out C# and Java? What efficiency impact does higher level languages give us and is it worth the performance loss? In this chapter we will try to discuss and understand what is important to consider when comparing programming languages with the help of the information collected from our tests. The results of our investigation will be linked to the work done by others before us in order to reach a conclusion about the current feasibility of running higher level languages in resource-constrained devices having similar performance as the devices investigated by us.

It is important to realize that our investigation is focused on what tomorrow's programming platform for these kind of embedded systems are. Tomorrow will bring more RAM, more CPU power and more flash memory. One of the reasons behind this thesis work is that we have a belief that many embedded systems have or will in the near future reach a point were a transition to higher level languages will be feasible.

Another important aspect to consider when reading this section is that as previously noted in section 1.3, we say that we measure the performance of the languages themselves, but in reality we are looking at specific virtual machine implementations for these languages.

5.1 Performance of the Languages

The performance results illustrated in chapter 4 provided us with a clean and comparable overview of the performance of the different languages in the Axis Communications camera platforms. The initial conclusion that can be gathered from them is that while they all differ in their data, the curves still all look very similar to each other. This strengthens our claim that while the programs are all written independently and not based on each other, they still solve the same task since the CPU load and memory usage patterns are so similar. That being said, it was still we who wrote all of the different versions of the programs, and we always had in mind to use the same algorithms and techniques when possible in the different languages, even though the library classes and functions differed. This gives us additional strength in our claim that these different programs are comparable in terms performance. An alternative study could have been to do it as in [7] where many programmers solved the same problems, and then an interval of the worst to best performing implementations were presented in the results. This type of study naturally has a wider difference between the implementations compared to the programs completely written by us which had the aim to be as similar as possible while still being implemented using different libraries.

5.1.1 Execution Time

In general, the C programs performed better than Java and C#. Though that does not mean that Java and C# performed badly. The actual execution time and memory usage were in some cases not very far behind that of the C programs. Take for example the result of the XML DOM parsers. The execution time only differed with about 3 seconds slower for C#. And Java being about 7 seconds behind C. Remember that this test is rather extreme due to parsing an XML file with a large size of about 15 megabytes. This is a lot of data to parse and in practice these documents tend to be much smaller. In a more realistic case of perhaps a couple of kilobytes of XML data, all of the languages would have completed the task without much difference at all. As mentioned before, we are dealing with a soft real-time system which means that a few seconds here and there may sometimes be an acceptable performance loss. Very similar execution time differences could be found in the SAX XML parsers as well as the snapshot programs. One interesting point to note is that the snapshot program was the only program where Java managed to outperform C# in all performance aspects. Java brought bad results in the motion detection test. Due to the big difference, we interpret this as cause of an implementation difference. In motion detection, floating point numbers are often used and since, as previously mentioned, the JVM was compiled to run with software floating point number calculations, this can partly contribute to the long execution time. The Java program could not handle processing all the images in a reasonable time if we compare it to C and C#. C# finished in about 48 seconds which is reasonable, compared to Java, but about 5 times longer time than C. As before, C outperforms both C# and Java.

5.1.2 CPU Usage

The CPU usage of the programs also presented us with some interesting information. For the XML parsers, all of the programs pretty much maxed out the CPU until the program finished executing. This is the expected output as the programs are never in a waiting state and they always perform some kind of calculations. However for the snapshot programs we saw a clear difference in CPU usage between the languages. C# started out by taking full use of the CPU for the first few seconds, and Java in a similar way but using a bit less CPU. C performed very nicely with a somewhat constant CPU usage at around 20-30% until execution of the program was finished. In contrast, both Java and C# saw a reduced use of CPU over their execution times. This might be explained by the fact that since the program repeats the same task ten times over, the JIT functionality of the virtual machines is used to optimize the calculations done after a few iterations [49].

5.1.3 RAM Usage

The RAM consumption between the different languages contained the largest disparities. Though this does not mean that the difference was large in all cases. For the XML DOM parser, C performed the best as usual, but C# was not far behind and managed to complete the task using only about 15 megabytes more memory. Included in this difference is the whole Mono runtime with a virtual machine, garbage collector and AOT/JIT mechanism. Considering all that then C# performs really well. The same can not be said for Java though, which uses a up to about 130 megabytes of RAM and thus uses about 100 megabytes more RAM than the C program. This could be related to the standard Java library's implementation of DOM parsing needing more RAM than the others. It could also be the JIT that stores a lot of native code in memory in the case of Java. If this the case then we can see significant differences in the AOT/JIT implementations between the JVM and Mono, since Mono does not show the same behavior. If we compare 130 megabytes of consumed RAM to the total amount available on our device (256 MB), we quickly realize that the amount is unacceptably much, but then again - who knows if 130 megabytes is a lot for a program on an embedded device, such as ours, in a few years?

For the XML SAX parser the memory usages were very similar. Important to note is that in the C program it was up to the programmer to choose how much memory will be allocated at a time for the input XML file, since only part of it is loaded into memory while parsing. We purposely made this size similar to Java and C# by looking at the typical size allocated by those environments in `hogclient` (excluding the runtime overhead). The memory used by Java and C# was around 12 megabytes respectively. While we did not know the exact size of the memory used by the runtime environment, it was estimated, with the help of the numbers from `hogclient` and API calls from the `Runtime` class in Java and the `Process` class in C#, to be around 10 megabytes in this example. If the runtime size is not considered, the allocated memory from the actual program was not too far off from C. This is a good demonstration of the fact that smaller programs will always have a relatively much larger memory footprint in Java and C# compared to equivalent C programs, since the runtimes will always be present.

In order to eliminate the overhead of spawning a new instance of a runtime every time a new program needs to be started, the Unix system call `fork` can be used. A program would then solely be responsible for spawning other processes which means that they run in the same runtime.

In order to tweak the behavior of the managed memory in Java, one can experiment with different garbage collectors. C4, ParallelGC and Dynamic Garbage Collector are three examples of different garbage collectors [21]. For Mono, we had the choice of two different garbage collectors, *sgen* and *boehm*.

5.1.4 Size on Secondary Memory

From our results, we can see a trend where compiled C code takes more memory space than CIL code for C# and Java bytecode for Java. The compiled C code in our examples was 3-10 times larger than the compiled CIL code and Java bytecode. Although, in the motion detection case, C at `263 kB` was the smaller than both C# and Java, at `346 kB` and `2800 kB`. Here, the code we wrote ourselves only present a very small subset of the actual compiled code. *AForge.NET*, which was used for C#, and *webcam-capture*, which was used for Java, took a lot more memory space than *vision*, which was used for C. Since only a subset of the frameworks were used, smarter techniques for excluding unused methods could have been used with for example `Apache Maven Compiler Plugin` for Java [50].

Java class files are smaller than executable C code due to the fact that bytecode instructions are one byte long compared to a machine code instruction that can be 2-8 bytes long [51]. D. Rayside, E. Mamas and E. Hons state that trying to optimize size on secondary memory for compiled Java class files is of great interest. They suggest that if Java should survive to embedded platforms, the size needs to be kept down [52].

5.1.5 Size of Runtime Environments

As mentioned before, compiled C programs can be run natively which means that we do not have to install a runtime after a clean install on the camera. However, when using C# and Java, the size of the runtime environment is significant in size.

For C#, we had a whopping size of 105.5MB. From Mono's website on Small Footprint, it says: *"The most basic mono install currently takes about 3.7 MB of disk space, this includes about 1.7 MB for the JIT and 2 MB for mscorlib.dll."* [53]. Our compilation of Mono was not optimized fully and therefore, the size might be misleading. It is also important to note that we include all standard libraries in order to have a full install, which of course is going to increase the size drastically.

There are a lot of method and compile parameters to provide in order to make Mono smaller in size, some of them that we have not tried. See the installation guide on GitHub for some of these [44]. Since Axis Communications also needed to include libgdiplus together with its dependencies (total of 26 MB), we have a larger runtime than in some other cases.

In the case of Java where we had the binary with libraries provided by Oracle, we had three different setups of configurations [54]. They were ordered from smallest to largest in size spanning between 14 - 61.4 MB where a larger configuration included the classes included in the smaller configuration. The configurations are simply cherry picked classes and can be manually chosen after the programmer's needs. The process of cherry picking libraries for usage can be a tedious process which means that a tool that manages this is needed, or the environment will only grow larger and larger. Even if there would be such a tool, the requirement for each application is different and the tool would have to be run continuously during the development process. Having a rich standard library has its price, memory usage.

It is hard to say if the runtime environments take too much space or not. Our camera has, as previously mentioned, 128 MB of flash memory. We can not afford to use too much of it. But how much is too much? What happens in the future, what if the next camera in the same series has 256 MB flash memory? Things that were too large in the previous camera might not be too large in this one. Without too much knowledge in the absolute requirements at Axis, we think that we can squeeze in Java. If Mono would be as small, then it could also work. Although, work would have to be done to reduce the size.

5.1.6 The Overall Performance

A result found in [7] was that scripting languages consumed about twice as much memory as the C and C++ equivalents. Furthermore, Java was interestingly enough found to use even more memory than the scripting languages, using about three to four times as much memory as the C and C++ equivalents. The results in the article, just like we do, include the Java runtime system in their measurements. Looking at the data in the article it is seen that the program memory only required a few megabytes of memory for C implementations, this is most likely why the Java implementation can use up to four times as much memory. Just like we found in our tests, programs with a small implementation size will create a large difference in memory due to the size of the Java runtime. If the test in the article would have used a more memory consuming task, the difference would not have been as large, as seen in our XML DOM programs.

It is important to notice that for Java, we only had floating point number calculations in software instead of hardware which very well could have had an impact on the results.

An interesting observation in [7] was that scripting languages used less memory than Java. Which might make the reader think that it would have been suitable to test these scripting languages on our investigated embedded systems. We initially thought that scripting languages would be too slow for our causes due to them mostly being interpreted. A result in the article strengthens that thought by stating that while scripting languages used less memory than Java, they also performed about 10 times slower than C and C++, while Java only was about 4 times as slow. Even though we see these numbers, we still see scripting languages in embedded systems as a very interesting area for further research.

5.2 Programmer Productivity

A language/environment in which close to perfect programs can be produced does not necessarily need to be a good language/environment. Even if all the C programs executed in shorter times compared to C# and Java, the time it took to write the program was significantly longer. For the snapshot program in C, we had to write the code for doing an MD5 hash for the authentication process and even code for converting strings to Base64. Because of this tedious process, we chose to include an external library, libcurl [55], instead of writing it completely ourselves with just the standard library.

There are different opinions on whether a simple text editor is easier to use than an IDE. An employee at Axis Communications mentioned that he felt much more productive if he could use Visual Studio when programming in C# than a normal text editor. In a blog post, R. Anklam mentions something important when it comes to programmer productivity. [56]:

> "In conclusion I'd like to say that no text editor or IDE will make every developer happy. Each developer should use whatever makes them happy and productive."

5.2.1 Language Functionality

As we can see in section 2.5, taking in an advanced runtime can bring a lot of useful features. Bigger things such as object oriented programming, generics, exception handling and garbage collection can greatly increase programmer productivity and is therefore a good argument for bringing such a platform on board [57]. Even smaller things such as reflection, lambda functions and operator overloading are things to be considered.

5.2.2 Our Experiences

In this section we will discuss our experiences when writing the test programs in the different languages. General discussions and results from other work will also be taken into consideration and compared to our experiences. Since we did not have the resources to conduct an extensive study by for example letting employees at Axis Communications implement the test programs in this report, we will instead discuss our personal experiences of writing these programs.

Initially we set out to implement the snapshot program with the idea of not using any libraries aside from the languages' standard libraries. The reason we wanted to do this method was to show how much effort was needed to create these tools without relying on third party libraries. However, we soon ditched this idea due to the C implementation becoming too complex. This was due to the fact that we needed HTTP and FTP functionality in the program, and using C without third party libraries required us to communicate at a low level using these protocols. This in itself is a good demonstration of why programmer efficiency is greatly reduced in some situation when writing C code. While our problem could be solved by using the third party library libcurl which has a permissive license, it may not always be the case. Some companies have stricter policies when it comes to bringing in third party libraries than just looking and the license and give a thumbs up. Besides, having to depend on too many third party libraries greatly increases the risk of dependency problems down the road. There is also always the time and effort needed to look up third party alternatives in order to solve tasks like this at a higher level. Having functionality like HTTP, XML and so on in a standard library can thus save much time in the development process.

While the idea of not using third party libraries was ditched, we still wrote the low level HTTP communication in order to be able to compare it to the higher level solutions. In order to create the simple HTTP connection to a web server with digest authentication with the help of only standard libraries in Java and C#, the total amount of lines of code were around 10-30. In C, we had 200 lines of code. Whilst lines of code is a bad measurement for comparing programming languages, it still says something about how much a programmer has to write, as mentioned earlier in the report. You would preferably want to write as few, while easily readable, lines of code as you can. We found that for the C solution, much of the code we had to write and spent time to get working was related to string manipulation. Managing the strings in the C solutions required a significant amount of effort as opposed to the higher level languages. In Java, all that was needed was to provide the parameters needed (URL, user name, password), open and read, and then close the stream.

In a perfect scenario, we would have liked to have multiple groups developing the same, larger scaled application in different environments instead of doing smaller examples as we had to restrict us to. Obviously, these kind of resources were not available and instead we had to make a downscaled version of it. We are sure that some points that would be discovered with a more sophisticated process were not found at all in our tests.

5.3 Choosing a Language for Embedded Systems Development

An important decision for system designers is to choose which language will be the most suitable for the task. As we have seen in this thesis, the choice involves a trade off between programmer productivity and performance. In order to make this decision some core fundamentals need to be considered. In [1] the most important factors when choosing a language for an embedded system was described as the following:

- Due to limited resources, the language must be efficient enough to meet the system constraints.
- Embedded systems require low-level access to hardware, so the language need constructs to do this, for example with pointers.
- The language must be able to support flexible libraries that can be easily ported to different architectures.
- The language must be widely used in order for the developers to be able to recruit new experienced developers, and to guarantee that there are sufficient learning materials available.

Using these factors as a base for our analysis, we argue that C# is satisfying all of them to a great extent. The only factor of doubt is the performance requirement, which has been one of the investigated aspects in this thesis. For Java it is a bit different since there is no low-level access to hardware without using JNI for calling C functions from Java code.

Something the above factors does not consider is the ability to use multiple languages on an embedded device. If this is the case then Java does not necessarily pose a problem due to its lack of low-level hardware access, since those parts could be written in C. This concept is actually of great interest for embedded devices strong enough to handle it, since it allows the developers to program drivers and resource heavy tasks in C, while keeping the typical programs written using a higher level language.

5.4 Other Important Aspects

Coming from a background of high-level languages, it is hard to imagine writing high-level applications without the basic fundamental parts of a high-level languages such as garbage collection or object oriented programming. Research has found that the presence of object orientation helps representing the problem domain, which thus makes it easier to create good designs for software [1].

Seeing as both C# and Java are more popular than C today [6], we can with confidence say that it is much easier to find a C#-programmer or a Java programmer than it is to find a C programmer. Since C is more popular on lower levels of programming, many of the C programmers are skilled in those areas and not in the type of high-level applications that are developed at Axis Communications at the Firmware Applications department.

It is very important to realize that different programmers write different code and that can be a factor when it comes to performance difference. F. Pettersson and E. Westrup brings up an article written by L. Flon, where he says [13][58]:

> ”There does not now, nor will there ever, exist a programming language in which it is least bit hard to write bad programs.”

Well, what repercussions does that statement imply? First of all, our written test programs are written by two different people; Daniel and Philip. Second of all, even if the programs were to be written by only one of us, there is nothing that can guarantee that programs between languages are equivalent or that it is just to compare them. We tried to battle this problem by writing different use cases instead of just one.

5.5 Future

What happens in the future? Perhaps someone will write a master thesis on going from C# or Java to another new fancy programming language. The programming world moves fast and systems constantly need to be reevaluated.

Perhaps high-level languages will be standard in the future on embedded systems. Maybe the shift we have had with people writing in assembly code to C code will happen again but with C to Java. Microsoft and Red Hat entered a partnership focusing on making the .NET platform work on Linux [59]. In the beginning of 2016, Microsoft bought Xamarin, the company behind Mono [60]. Microsoft is realizing the importance of cross-platform, and functionality - perhaps even for platforms with constrained resources such as embedded systems. If Mono can function on an embedded system under the wings of Xamarin, just imagine what Microsoft could do with the resources they possess.

Chapter 6
Summary

We do see a strong development in the field of higher levels of programming abstraction on embedded devices. We can see that this does actually enable for many use cases on a higher level.

6.1 Conclusion

Did we answer our question?

> Have modern embedded devices, such as cameras at Axis Communications, reached a point where it is feasible and advantageous to transition to higher level languages instead of C?

Yes, they have. At least for some applications, we can actually see that a transition to high-level languages is feasible. We believe that such a switch could, with the help of the tools offered by the high level language platforms, greatly improve the development time of certain applications. More specifically, applications that do not require maximal performance such as motion detection, but instead applications which requires less performance. Some examples are HTTP communication, parsing and file manipulation. Obviously, both use cases are hardware differ and therefore, there is not one simple, correct answer to the question that covers all kinds of modern embedded devices. We only scratched the surface of this giant question. In order to investigate the problem further with all kinds of optimization for all different environments, a lot of know-how is required.

6.1.1 Recommendation to Axis Communications

We recommend that Axis Communications and specifically the employees at Firmware Platform: Firmware Applications to give C# together with Mono a chance for high-level applications. Seeing as this needs to be tested for larger projects we think it would be necessary for Axis Communications themselves to try it out in a real project. It is necessary when reviewing a programming language [13, p. 49]. This is also due to the fact that project-based learning is a technique which highly encourages a learner because it presents challenges which have to be solved. This enables a fast learning process [61]. Mono requires a larger runtime than required today but we believe that the power the environment comes with can accelerate development. There is a lot of room for improvement and it needs to be investigated further, preferably employees with experience in build systems.

6.2 Future Work

In this section some aspects that were not studied in our thesis but still directly related to the work we did will be presented. This is partly a recommendation to Axis about what they should look into before making practical use of the results of our work. We will also present some more general future work relating to the field of study of this thesis.

6.2.1 Further Investigation for Axis Communications

In this thesis we have focused on getting Mono and the JVM to run on the target platforms used at Axis Communications. What we have not done is a thorough practical investigation of the optimal methods to get the languages well integrated into the company ecosystem. Before any actual project is done we would suggest Axis Communication looks into the following problems:

- Integrate the Mono source code along with the external (but important) library `libgdiplus` and its dependencies into the platform SDK by constructing necessary makefiles.
- Further analyze optimization flags at compile time to reduce size of the binaries. It is interesting to compile with the `-mthumb` flag for ARM processors in order to take advantage of the 2 byte instruction set *Thumb*, available in ARM processors [62]. The default instruction set uses 4 bytes per instruction, so this could in the best case reduce the binary size in half. For some machines and programs, this results in increased performance due to the fact that the instructions are shorter. In other cases, for example using 32 bit integers, it becomes slower. Here we have to retrieve an integer with multiple instructions.
- Strip all binaries to save space, using the available toolchains.
- Investigate which C# classes in the standard library that are of interest to the company, in order to selectively only include those in the SDK in order to save space.
- Try out the low level hardware access through the `unsafe` keyword available in C#.

6.2.2 Unexplored areas

Our study was done at a specific company using their products to carry out the investigation. It is interesting to look at use cases like this to get real world results in the field of study of the performance of higher level programming languages in resource-constrained embedded devices. Our results were overall positive but there are some future work that could clear up some questions that were excluded in our investigation:

- Our results and discussion indicated that a partial transition from C to a higher level language such as C# or Java is a very probable alternative for the developers at Firmware Applications at Axis Communications. However, a full transition would most likely need further study with more powerful devices, perhaps a couple of years in the future. In a similar way, less powerful devices that are not necessarily related to Axis Communications need to be studied in order to find the limit of where it stops being feasible to run C# or Java on the devices.

- Investigation of scripting languages were excluded in this work due to the fact that they were, considered too resource demanding, see subsection 5.1.6. It would be interesting to do an investigation similar to the one in this thesis in order to find out more about the feasibility of running them as well. This is especially interesting due their lower memory footprint as discussed in subsection 5.1.6 as well.

Bibliography

[1] M. Nahas and A. Maaita, *Choosing Appropriate Programming Language to Implement Software for Real-Time Resource-Constrained Embedded Systems*. INTECH Open Access Publisher, 2012.

[2] M. H. Lutz and P. A. Laplante, "IEEE Software: Real-Time Systems - C# and the .NET Framework: Ready for Real Time?" *IEEE Distributed Systems Online*, vol. 4, no. 2, 2003.

[3] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010* (C. Morin and G. Muller, eds.), pp. 69–82, ACM, 2010.

[4] "RTSJ Main Page." `http://www.rtsj.org/`. Accessed: 2016-04-04.

[5] E. Bertolissi and C. Preece, "Java in real-time applications," *Nuclear Science, IEEE Transactions on*, vol. 45, pp. 1965–1972, Aug 1998.

[6] Nick Diakopoulos and Stephen Cass, "Interactive: The Top Programming Languages 2015," *IEEE Spectrum*, 2015.

[7] L. Prechelt, "An Empirical Comparison of Seven Programming Languages," *IEEE Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[8] C. Jones, "Software metrics: Good, bad and missing," *Computer*, vol. 27, no. 9, pp. 98–100, 1994.

[9] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[10] G. D. Smet, "How much faster is Java 8?" `http://www.optaplanner.org/blog/2014/03/20/HowMuchFasterIsJava8.html`, 2014. Accessed: 2016-03-03.

[11] "The Computer Language Benchmarks Game." `http://benchmarksgame.alioth.debian.org/`. Accessed: 2016-03-08.

[12] "So, why measure toy benchmark programs?" `http://benchmarksgame.alioth.debian.org/why-measure-toy-benchmark-programs.html`. Accessed: 2016-03-08.

[13] F. Pettersson and E. Westrup, "Using the Go Programming Language in Practice," Master's thesis, Faculty of Engineering LTH, 2014.

[14] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. USA: Addison-Wesley Educational Publishers Inc, 4th ed., 2009.

[15] "Embedded Computer Systems - Department of Electrical and Computer Engineering." `http://www.ece.ncsu.edu/research/cas/ecs`. Accessed: 2016-03-02.

[16] "Real-Time Systems and Programming Languages - Chapter 1." `https://www.cs.york.ac.uk/rts/books/RTSbookThirdEdition/chap1.pdf`. Accessed: 2016-03-03.

[17] D. Bindel, "Lecture 19: High-level languages and Julia." April 2014.

[18] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth, "Fast, effective code generation in a just-in-time java compiler," *SIGPLAN Not.*, vol. 33, pp. 280–290, May 1998.

[19] "Creating a memory leak with Java." `http://stackoverflow.com/questions/6470651/creating-a-memory-leak-with-java`. Accessed: 2016-03-08.

[20] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pp. 151–160, May 2008.

[21] "Azul C4 Garbage Collector - Azul Systems, Inc. Azul Systems, Inc.." `https://www.azul.com/resources/azul-technology/azul-c4-garbage-collector/`. Accessed: 2016-04-01.

[22] B. Venners, "Bytecode basics: A first look at the bytecodes of the Java virtual machine," *Javaworld:`http://www.javaworld.com/jw-09-1996/jw-09-bytecodes.html`*, 1996.

[23] M. Felleisen, "On the expressive power of programming languages," *Sci. Comput. Program.*, vol. 17, no. 1-3, pp. 35–75, 1991.

[24] "Introduction to Static and Dynamic Typing." `http://www.sitepoint.com/typing-versus-dynamic-typing/`. Accessed: 2016-03-02.

[25] "Is C# a strongly typed or a weakly typed language?" https://blogs.msdn.microsoft.com/ericlippert/2012/10/15/is-c-a-strongly-typed-or-a-weakly-typed-language/. Accessed: 2016-03-02.

[26] "Type Inference." https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html. Accessed: 2016-03-22.

[27] "C Lambda expressions: Why should I use them?" http://stackoverflow.com/a/167363/5513627. Accessed: 2016-03-09.

[28] "Computer Architecture: Why did the x86 architecture become more popular than the MIPS architecture?" https://www.quora.com/Computer-Architecture/Why-did-the-x86-architecture-become-more-popular-than-the-MIPS-architecture. Accessed: 2016-03-10.

[29] "What "Worse is Better vs The Right Thing" is really about." http://yosefk.com/blog/what-worse-is-better-vs-the-right-thing-is-really-about.html. Accessed: 2016-03-10.

[30] "file." http://pubs.opengroup.org/onlinepubs/9699919799/utilities/file.html. Accessed: 2016-03-10.

[31] "Scratchbox." http://www.scratchbox.org/. Accessed: 2016-03-10.

[32] "AXIS M30 Network Camera Series | Axis Communications." http://www.axis.com/global/en/products/axis-m30-series. Accessed: 2016-03-22.

[33] "Axis introduces high performance mini dome cameras for discreet indoor video surveillance | Axis Communications." http://www.axis.com/se/sv/press-center/press-release/4248. Accessed: 2016-03-23.

[34] "Ambarella | HD and UHD Video Processing." http://www.ambarella.com/news/54/122/Ambarella-S2L-IP-Camera-SoC-family-enables-a-new-generation-of-intelligent-security-cameras. Accessed: 2016-03-22.

[35] "Cortex-A9 Processor - ARM." http://www.arm.com/cortex-a9.php. Accessed: 2016-03-22.

[36] "GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)." https://gcc.gnu.org/. Accessed: 2016-03-02.

[37] "Mono Project." http://www.mono-project.com/. Accessed: 2016-02-03.

[38] J. King and M. Easton, *Cross-platform. NET Development: Using Mono, Portable. NET, and Microsoft. NET*. Apress, 2004.

[39] "libgdiplus." http://www.mono-project.com/docs/gui/libgdiplus/. Accessed: 2016-03-03.

[40] “Java SE Development Kit 8 - Downloads.” http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. Accessed: 2016-03-02.

[41] J. Kuhner, *Expert. NET Micro Framework.* Apress, 2009.

[42] E. Rubin and R. Yates, *Microsoft. NET Compact Framework: Kick Start.* Sams Publishing, 2003.

[43] “Compatibility.” http://www.mono-project.com/docs/about-mono/compatibility/. Accessed: 2016-03-03.

[44] “Code snippets from our Master Thesis at Axis Communications.” https://github.com/dabbe/msc-thesis-code. Accessed: 2016-03-24.

[45] R. W. Scheifler and J. Gettys, “The X window system,” *ACM Transactions on Graphics (TOG)*, vol. 5, no. 2, pp. 79–109, 1986.

[46] “RAPP User’s Manual.” http://www.nongnu.org/rapp/doc/rapp/. Accessed: 2016-03-02.

[47] “AForge.NET :: Computer Vision, Artificial Intelligence, Robotics.” http://www.aforgenet.com/. Accessed: 2016-04-05.

[48] “Webcam Capture in Java.” http://webcam-capture.sarxos.pl/. Accessed: 2016-03-21.

[49] M. Paleczny, C. Vick, and C. Click, “The java hotspot TM server compiler,” in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pp. 1–1, USENIX Association, 2001.

[50] “Apache Maven Compiler Plugin.” https://maven.apache.org/plugins/maven-compiler-plugin/. Accessed: 2016-03-08.

[51] V. D. MAREDU, P. PARVATANENI, V. K. R. MANGIPUDI, and C. AVASARALA, “Optimizing Java-to-C Compiler for Embedded Systems,” *Atti Della Fondazione Giorgio Ronchi Anno LXII N. 6*, p. 759.

[52] D. Rayside, E. Mamas, and E. Hons, “Compact Java Binaries for Embedded Systems,” in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON ’99, pp. 9–, IBM Press, 1999.

[53] “Small footprint | Mono.” http://www.mono-project.com/docs/compiling-mono/small-footprint/. Accessed: 2016-03-24.

[54] “An Introduction to Java 8 Compact Profiles (Jim Connors’ Weblog).” https://blogs.oracle.com/jtc/entry/a_first_look_at_compact/. Accessed: 2016-03-24.

[55] “libcurl.” https://curl.haxx.se/libcurl/. Accessed: 2016-03-04.

[56] "The Great IDE vs Text Editor Debate: Why I Switched Sides." `http://blog.bittersweetryan.com/2012/02/great-ide-vs-text-editor-debate-why-i.html`. Accessed: 2016-03-09.

[57] S. McConnell, *Code complete*. Pearson Education, 2004.

[58] L. Flon, "On Research in Structured Programming," *SIGPLAN Not.*, vol. 10, pp. 16–17, Oct. 1975.

[59] "Red Hat and Microsoft making .NET on Linux work for Enterprises." `http://developerblog.redhat.com/2015/11/04/red-hat-microsoft-making-dot-net-on-linux-for-enterprises/`. Accessed: 2016-03-09.

[60] "Microsoft Buys Xamarin (At Last)." `http://fortune.com/2016/02/24/microsoft-buys-xamarin/`. Accessed: 2016-03-02.

[61] P. C. Blumenfeld, E. Soloway, R. W. Marx, J. S. Krajcik, M. Guzdial, and A. Palincsar, "Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning," *Educational Psychologist*, vol. 26, no. 3-4, pp. 369–398, 1991.

[62] X. Xu, C. T. Clarke, and S. R. Jones, "High performance code compression architecture for the embedded ARM/Thumb processor," in *Proceedings of the 1st conference on Computing frontiers*, pp. 451–456, ACM, 2004.

EXAMENSARBETE VID INSTITUTIONEN FÖR DATAVETENSKAP PRESENTERAT DEN 28 APRIL 2016

Högnivåspråk i inbyggda system

Författare/Authors: Philip Mårtensson & Daniel Olsson
Handledare/Supervisor: Assad Obaid (Axis Communications), Patrik Persson (LTH)
Examinator/Examiner: Krzysztof Kuchcinski (LTH)

POPULÄRVETENSKAPLIG SAMMANFATTNING

Varför används fortfarande C som huvudspråk i inbyggda system? Är inte dagens inbyggda system så pass starka att de klarar av populära högnivåspråk så som C# och Java, och på så sätt fly vissa av de problem som inbyggda system innebär?

I de fall där det inte krävs alldeles för mycket prestanda i ett scenario med ett inbyggt system, är en övergång till ett högnivåspråk definitivt lämplig.

Introduktion

Användningen av språk på högre nivå såsom C# och Java ger en programmerare många verktyg för att snabbt och enkelt kunna producera fungerande kod som är lätt att underhålla. Trots detta kör nästintill alla inbyggda system idag språk på en lägre nivå såsom C. Detta har tidigare berott på att miljöerna som behövs för att köra C# och Java ofta kräver mer av enheten än vad C gör. För varje dag som går blir hårdvaran som behövs i många inbyggda system bara billigare och billigare. Hur mycket krävs det utav ett system för att köra högnivåspråken och när är prestandaförlusterna för stora för att kunna använda sig utav ett högnivåspråk?

Effektivitet hos programmeraren

Användningen utav högnivåspråk har visat sig flertalet gånger ge en programmerare högre kvalitet på sin kod och dessutom en ökad produktivitet. Dessutom har det visat sig att tiden det tar att skriva program förkortas, vilket är det som företag prioriterar högst. Detta är anledningen till att högnivåspråken är attraktiva att använda.

Undersökning

För att svara på alla dessa tidigare ställda frågor om hur ett skift från C till C# eller Java skulle hjälpa programmeraren gjordes det en undersökning i samtliga språk. Ett par olika program, typiska för Axis Communications, skrevs i alla tre språk och för alla tre gjordes jämförelser. Resultaten bestod av hur mycket internminne som användes under körtiden, hur mycket processorn användes samt hur lång tid det tog att köra programmet. Denna prestanda diskuterades tillsammans med andra fördelar med att använda högnivåspråk för att nå en slutsats om det var lämpligt att övergå till högnivåspråk delvis, fullständigt eller inte alls. I arbetet diskuterades även vad som definieras som effektivitet och produktivitet hos programmerare. Utöver detta gjordes en teoretisk jämförelse mellan alla de tre programspråken.

Slutsats

För att vi skulle kunna dra en vettig slutsats togs det hänsyn till de mätbara resultaten, teoretiska skillnader, såsom unika tekniker i språken, samt mått som är svårare att mäta. Som tidigare nämnt är slutsatsen att man kan spara in effektivitet om det görs ett skifte från C till ett språk på högre nivå såsom C# och Java för de program där prestandan inte är kritisk. För programmen med de mer kritiska kraven bör C fortfarande användas men i framtiden kommer man nog se en utveckling här också där man kommer kunna använda sig utav språk såsom C# eller Java.