NTNU

Norwegian University of Science and Technology

FPGA Development in The Cloud Using The IDE8 Developer Framework

Kristian Aalde

Master of Science in Electronics

Submission date:	June 2018
Supervisor:	Per Gunnar Kjeldsberg, IES
Co-supervisor:	Joar Rusten, Trådløse trondheim

Norwegian University of Science and Technology
Department of Electronic Systems

Master Thesis Assignment

Candidate name:

Kristian Aalde

Assignment title:

FPGA development in the cloud using the IDE8 developer framework

Assignment text:

This assignment shall focus on designing an FPGA stack for IDE8. IDE8 (https://ide8.io) is a generic web based development environment that can be customized for different programming languages and paradigms.

IDE8 is composed of two main components: the Integrated Development Environment, and the IDE8 Agent. The IDE8 Agent enables development boards to connect to the Internet where they can be picked up by IDE8.

As part of the assignment the student shall:

Study state-of-the-art FPGA development tools and methodologies with particular focus on how interfaces between tools can be implemented.

Get an overview of the IDE8 developers framework and the IDE8 API extension points, and discuss their suitability for an FPGA stack.

Get an overview of the IDE8 Agent and discuss its suitability with respect to netlist programming, debugging and instrumentation of an FPGA.

Discuss which tools are required in an FPGA stack in IDE8.

Build a working, proof of concept, FPGA stack with basic functionality including programming a suitable FPGA via the IDE8 Agent, and evaluate the proposed concept with ease of use, resource requirements, and scalability in mind.

Assignment proposer:

Joar Rusten at Trådløse Trondheim

Supervisor:

Per Gunnar Kjeldsberg

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Electrical Engineering Graduate Student

by

Cloud computing has become a large industry over the last decade. Furthermore, companies like Amazon Web Services (AWS) has begun to integrate FPGAs into their cloud infrastructure, allowing customers to build software applications that utilize FPGAs to accelerate computations. However, with regards to FPGA development, they only offer the Xilinx tool Vivado, and the development experience is very similar to traditional local development. Furthermore, because the FPGA boards are physically integrated with the AWS servers, they are unsuited to use for a normal FPGA design process for embedded systems. IDE8 is a cloud solution that aims to provide a web-based development environment for electronics. Offering FPGA development on this platform can be a way of making FPGA development more accessible and efficient. In this thesis, it has been developed a prototype of an FPGA tool chain by utilizing Docker containers and open source FPGA tools, that can be deployed on the IDE8 platform. The tool chain utilizes the open source tools from Project IceStorm, which is a set of tools for performing synthesis, place-and-route, and bitfile generation on the Lattice iCE40 FPGA. The tool chain was built in three different ways, by being split up into 1, 2 and 3 containers. By splitting up the tool chain into multiple containers, the disk space required to store the Docker images could be reduced. This is because when the solution is scaled over multiple virtual machines (VM), splitting up the tool chain allows one to copy over only parts of the tool chain to the new VM. Furthermore, by utilizing multi-stage builds, the image size of the Arachne-pnr and IceStorm containers were reduced by 73% and 51% respectively. It was tested if the three solutions behaved differently given the same memory and CPU constraints, but no significant difference was detected. It was concluded that the best solution was the 3-container solution, as it provides much more flexibility with regards to adding and changing components in the tool chain later on and provided the most scaling benefits with regards to image size.

Furthermore, the AWS FPGA developer AMI (Amazon Machine Image) was tested in this thesis and compared to IDE8. It is clear that IDE8's web-based development environment is very different from the development environment provided by AWS. The web-based environment is a lot simpler to start using, as it does not require any infrastructure management. This also makes IDE8 much easier to use than AWS's IaaS solution, especially for small projects and for students.

The FPGA tool chain developed in this thesis was also integrated into the IDE8 environment. Tests were performed, showing that it functioned as intended. For simplicity, the 1-container architecture was used for this.

Preface

This thesis is in part based on a report I made in the fall of 2017 [1]. A few of the relevant pieces of background information provided in Chapter 2 and Chapter 3 in this thesis are fetched from that report. I would like to thank *Wireless Trondheim* for their help throughout this process, with special thanks to Joar, for helping me staking out a good path for this thesis and for advising me throughout the semester, and Jon Anders, for integrating the solution into the IDE8 platform and for providing technical tips and insight. Furthermore, I would like to thank my supervisor, Per Gunnar, for following up on the progress of this thesis work and providing insight that has (hopefully) helped me produce a good report. I hope that the work done in this thesis will be useful, and that the FPGA stack will be continued to be worked on and improved, as I genuinely believe that this could be a very good way of offering FPGA development tools and make FPGA development more accessible. To help making the further development process easier, the code has been made available on Bitbucket [2]. I apologize in advance for the messy file structure and code of the repository.

This has been a very interesting project to work on, and I have come across many new technologies and learned a lot. However, it has also at times been a cumbersome process, and finding good tools to get accurate measures of containers resource usage was a difficult task. I do feel however, that I have been able to get through, or at least around, many of the obstacles that I have faced, even though there still are other things that I had wished to investigate further.

Contents

List of Figures

List of Tables

Abbreviations

AFI	**A**mazon **F**PGA **I**mage
AMI	**A**mazon **M**achine **I**mage
API	**A**pplication **P**rogrammable **I**nterface
AWS	**A**mazon **W**eb **S**ervices
BLIF	**B**erkeley **L**ogic **I**nterchange **F**ormat
CPU	**C**entral **P**rocessing **U**nit
DMA	**D**irect **M**emory **A**ccess
EC2	**E**lastic **C**ompute **C**loud
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPU	**G**raphics **P**rocessing **U**nit
HDK	**H**ardware **D**evelopment **K**it
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh-**L**evel **S**ynthesis
IaaS	**I**nfrastructure **as a** **S**ervice
IDE	**I**ntegrated **D**evelopment **E**nvironment
LUT	**L**ook **U**p **T**able
OS	**O**perating **S**ystem
PaaS	**P**latform **as a** **S**ervice
PNR	**Place**-and-**R**oute
RTL	**R**egister **T**ransfer **L**evel
SaaS	**S**oftware **as a** **S**ervice
SDK	**S**oftware **D**evelopment **K**it
SO	**S**ecurity **O**bjective
VM	**V**irtual **M**achine

Chapter 1

Introduction

1.1 Motivation

In today's modern world, there is a constantly increasing need for computational power. Because of this, over the last ten years, cloud computing has become a major industry. At the same time, hardware designs are becoming more complex, where even creating and synthesizing fairly simple designs, require advanced tools and a lot of computational power. Recently, people have realized that utilizing the cloud for FPGA (Field Programmable Gate Array) development has a lot of potential. Furthermore, FPGAs have become useful in many new ways, particularly software acceleration, most noticeably in the form of AWS's EC2 F1 instances (Amazon Web Services Elastic Compute Cloud FPGA instances). However, there are still many unexplored possibilities of utilizing the cloud for hardware development.

The cloud offers virtually unlimited amounts of computational power at a fairly low cost. The cloud would hence be a great environment for running the advanced and heavy syntheses and simulations that are required in modern FPGA development. The cloud is also a great environment for integrating and bundling services together. Hence, integrating project management tools and other tools with the hardware development tools is a possible future feature of cloud-based FPGA development. The cloud's pay-as-you-go payment model, where one is only charged for the actual usage of a service, will contribute to break down the cost barriers of starting a new hardware development project.

FPGAs are also likely to become more utilized in several different fields in the future. Compared to traditional software execution, FPGAs possess the ability to run tasks with a much higher degree of parallelism and much faster than normal CPUs (Central Processing Unit) and even GPUs (Graphics Processing Unit). The reprogramability of FPGAs also make them able to adapt to new tasks and to be upgraded with little to no additional costs. There is also being made great advances in tools for HLS (High-Level Synthesis), which makes it possible for software developers to write high level code like C or C++ to be executed on FPGAs.

Wireless Trondheim has a newly created cloud based development platform named IDE8, which aims to provide a development environment for electronics, that can be accessed via a web browser. Adding functionality for FPGA development into this platform would be a substantial addition to the platform. Docker is a technology that is used to create software containers, which is a way of running applications in an isolated and stable environment in the cloud, while still being efficient and scalable. Therefore, utilizing Docker containers to build an FPGA tool chain on the IDE8 platform can be a way of providing easy access to FPGA development tools, for a large number of people.

1.2 Objective, Limitations and Approach

This master thesis is partially based on a project assignment [1] that was completed during the fall of 2017. In that report, different cloud-based solutions were presented and discussed. This thesis aims to build on the findings from that report and build a working prototype of a cloud-based FPGA development stack. The objective of this thesis is also to assess how well the IDE8 developer's framework is suited for an FPGA development stack.

Since it has already been specified that the IDE8 cloud framework will be used in this project, this report will focus less on different general aspects of cloud computing as opposed to the report from the project assignment, but if there are certain parts of the IDE8 framework that should be changed to better suit the needs for an FPGA development flow, suggestions for how to improve this will be presented.

The practical work of this assignment will mainly consist of finding existing software for FPGA development, that preferably are open source, and link them together to take advantage of the cloud, to build the FPGA development stack. The goal is not to build new development tools from the ground up.

This thesis will primarily focus on the core functionality of an FPGA tool chain and less on extra features like debugging, waveform viewer and other visualization tools. Some additional features will be mentioned and discussed, but they will not be included in the FPGA tool chain that is developed in this thesis work.

It will mainly be looked into utilizing open source tools for this thesis. This is mainly due to the flexibility and freedom one often gets with open source software, which gives more room for experiments and modification. This thesis will however, also look into the state-of-the-art tools that are used today and are already available in the cloud. In particular, the Vivado tool offered by Xilinx, and how it is utilized in the AWS FPGA developer AMI (Amazon Machine Image) will be investigated.

1.3 Main Contributions

The main contributions of this thesis are:

Developing an FPGA tool chain using Docker containers. Three different architectures were created, and they were tested in terms of memory usage and CPU usage, along with the image size, to see which architecture performed the best. Also, several other qualitative aspects of the architectures were taken into consideration. It was then concluded what solution that would be the best suited for IDE8.

Evaluating the IDE8 cloud platform with regards to FPGA development and compare it to the AWS FPGA service and another proposed approach. To do this the AWS FPGA developer AMI was tested.

With a lot of help from the IDE8 team, the FPGA tool chain prototype was integrated with the IDE8 platform. This thesis explores some possibilities for further integration and present some possible ways of improving the FPGA tool chain in the future.

1.4 Report Structure

This report consists of the following chapters:

Chapter 2 contains explanations of all the concepts and background theory that is necessary to understand the work, choices, and discussions that have been done in this report.

Chapter 3 contains examples of work that has been done in the same field as this report, or work that can be relevant to use in this thesis. It also contains descriptions of work that this report heavily depends on, like the IDE8 framework.

Chapter 4 describes the implementation of the FPGA tool chain prototype. It presents different architectures, shows how they were implemented and presents results obtained from testing the different architectures.

Chapter 5 is explaining how the tool chain from Chapter 4 is integrated with IDE8, and how it may be developed further. Furthermore, the AWS FPGA developer AMI is tested.

Chapter 6 further discusses and evaluates the solutions from chapter 4, both in the context of the results obtained from the simulations and in the context of how to most efficiently integrate the tool chain with IDE8. The IDE8 FPGA solution is also compared to the AWS FPGA solution and another proposed approach, which is referred to as the FPGA infrastructure solution in this thesis.

Chapter 7 concludes on which architecture is the most favorable, and how the IDE8 FPGA solutions stands compared to AWS and the other proposed solution. It also contains propositions for future work that can be done with the IDE8 FPGA tool chain.

Chapter 2

Background

This chapter will contain information about different concepts and technologies used in this project. Here, it will be described how the different concepts and technologies work, and what benefits and challenges there are when using them will be outlined.

2.1 Virtual Machines

A virtual machine (VM) is a concept introduced as early as in the 1960's. A Virtual machine can be defined as an efficient, isolated duplicate of a real machine [15]. That the VM is efficient, indicates that a VM is able to utilize a significant part of the host machines instruction set directly, which is different from for example an emulation or simulation of a machine [15]. In modern computer systems, a virtual machine is a software duplication of a physical machine, with its own virtual CPU, memory, storage, and OS (Operating System). The software used to create and manage virtual machines is called a hypervisor [16]. A hypervisor will have the ability to create several virtual machines running concurrently on the same physical machine. This ability is one of the key features that make virtual machines very useful in modern cloud computing, as it provides multiple isolated environments to run applications on the same hardware, which may lead to a much higher utilization of computer resources. Furthermore, it provides isolated environments for running applications, so they don't interfere with one another, and applications that are meant to run on different OS's may now run on the same hardware instance.

2.2 Containers

Containers are a different way of providing virtual and isolated computer environments compared to VMs. While a VM creates an entire machine with a complete OS running on top of it, a container shares the OS with the host [17]. This approach is a lot more lightweight than using VMs, as each instance of a virtual machine requires a lot of overhead in the form of its own OS that can be several GB in size, which in turn requires a lot of memory to run. A container only requires a few MB of overhead to run in its simplest form, hence it is much simpler to run multiple containers at once compared to virtual machines.

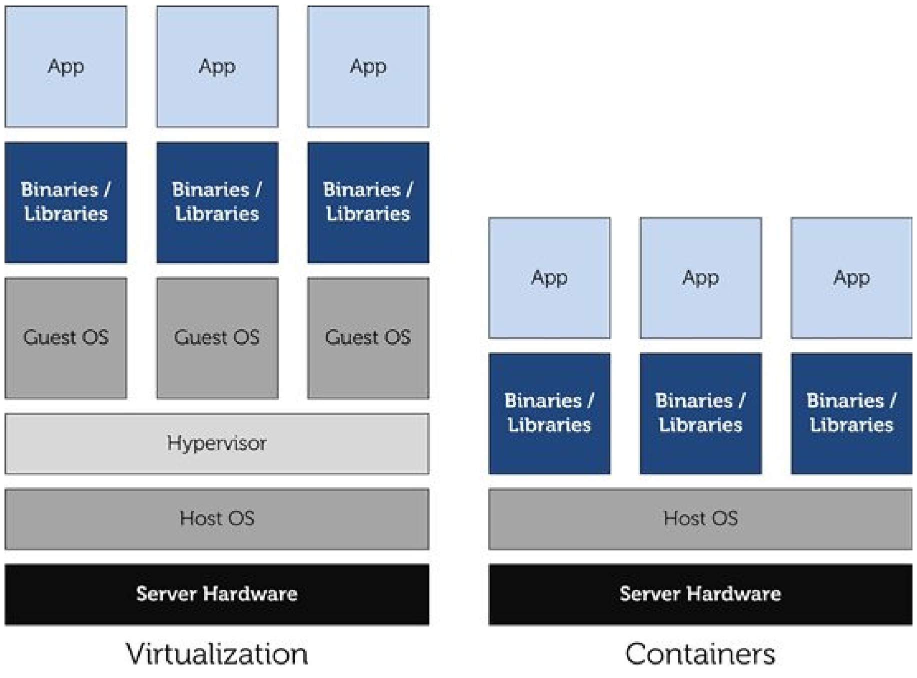


FIGURE 2.1: Comparison of VMs and containers from [3]

In Figure 2.1, a simple overview of the architectural differences between containers and VMs are shown. One can see that there are no hypervisor or guest OS needed for containers to operate. A container only consists of one or more applications and the binaries and libraries needed to run them, while utilizing the kernel of the host OS.

A container can also run inside a VM and utilize the kernel of the OS on the VM. Since containers are so lightweight on overhead and memory usage, it is also possible to run containers within a container.

2.3 Microservices

Microservices or "Microservice Architecture" describes a way of designing software applications as several smaller independent services [18]. There are few precise definitions that explains in detail what a microservice architecture is, but one simple way of defining it is [4]: "*At its simplest, the microservices design approach is about a decoupled federation of services, with independent changes to each, and agreed-upon standards for communication.*"

The traditional way of designing applications is referred to as "Monolithic Architecture" [4]. A monolithic design is typically divided into functional tiers, but the tiers are much more tightly coupled together than with a microservice architecture.

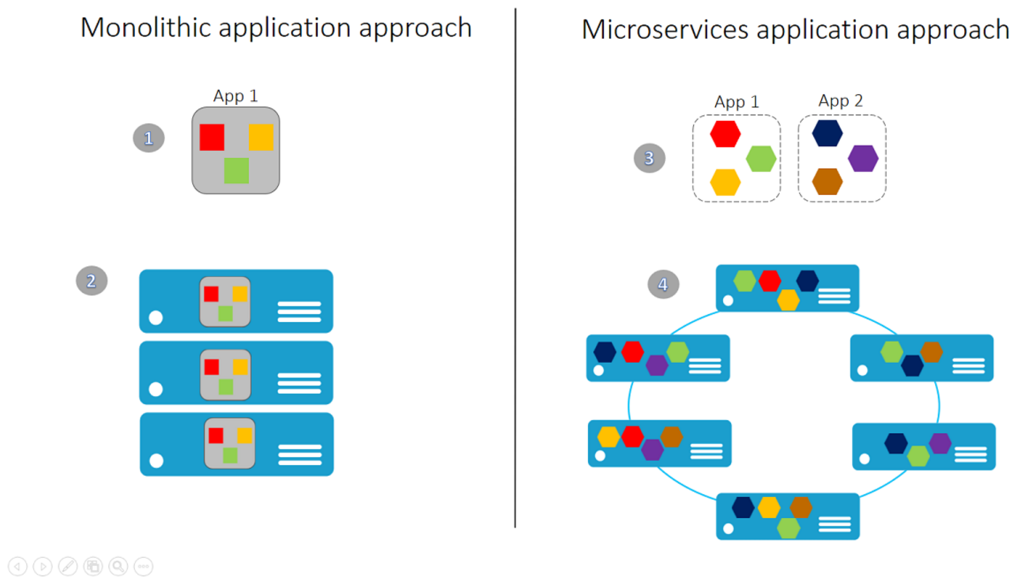

Monolithic application approach
Microservices application approach
App 1
App 1
App 2
1
2
3
4

FIGURE 2.2: Comparison of microservice and monolithic applications [4]

Figure 2.2 shows an overview of monolithic and microservice design and scaling. The figure has four different sub-figures:

1. In a monolithic application, there is domain-specific functionality. The application is usually divided into functional layers, such as web, business and data, but the different functionality is tightly coupled together.

2. A monolithic application is scaled by cloning it onto multiple servers, VMs or containers. The entire app must be cloned each time.

3. In a microsevice application, all functionality is separated into smaller services, that may run independently from each other.

4. A microservice application is scaled by deploying each service independently over different servers, VMs or containers. So, an application can be spread over multiple servers, VMs or containers.

There are many advantages with microservice architectures over monolithic architectures [4]. One of the greatest advantages is the improved scalability. As shown in Figure 2.2, when a monolithic application is scaled up, the entire application has to be cloned, but with a microservice architecture, each individual service can be copied independently from the other ones, this leads to much less overhead by ensuring that only the service that is needed is running. Another advantage with microservices, is the improved flexibility when developing and updating an application. Since all services run independently, each service can be updated and developed without affecting the other services in the application, as long as the communication interface between the services remain consistent. However, one disadvantage of a microservice architecture is this added need for communication between the different services, which may be significantly more complicated than with a monolithic architecture.

2.4 Cloud Computing

Today, instead of each company having to spend large amounts of resources on acquiring and maintaining their own servers and building platforms that connects its employees and customers to this hardware, several IT companies have specialized in providing these services to other companies and individuals. This solution, where companies rent out the computation power and storage, is called the cloud. There is no short and explicit way of defining what cloud computing is, but a part of The National Institute of Standards and Technology's (NIST) definition is [19]:

"*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*"

Cloud computing utilizes the benefits of operating on a large scale. Large cloud providers such as Microsoft and Amazon Web Services (AWS) own hundreds of thousands of servers that they have placed in data centers around the world.

Being a cloud provider is about more than just offering pure data storage and computational power. The cloud platform, that allows the users to connect and use the available resources in an efficient way, is just as important. Most cloud providers today also provide many different services within their cloud solution, such as machine learning software, big data processing services, and development platforms [20]. A cloud platform provides its resources to the user by utilizing virtualization, which were explained in Section 2.1 and 2.2. These platforms and techniques are really what separates a cloud from just a computer cluster.

2.4.1 History

The cloud computing concept was introduced as early as in the 1960s by John McCarthy [21, 22]. The history of why or how the word "cloud" came in to use is not clear though. In the early 1970s the first types of cloud computing became available. Companies could submit jobs to for example IBM and have them handle the computation. At that time computers were large and expensive, so for most companies it would have been difficult to acquire their own resources for computing. As computer technology improved, more and more businesses were able to acquire their own servers and computers. Even though work was still being done in the field of cloud computing, the modern form of cloud computing as we know today did not start to gain traction until the second half of the 2000s. Amazon released its Elastic Compute Cloud in 2006 [23], and in the following years, other competitors started to release their own cloud platforms.

2.4.2 Public Cloud

When people think about the cloud, most people probably think about the public cloud. This is when a company offers computer resources to customers and handles all the management of the physical servers. Customers typically connects to the cloud service via a web interface. The two major cloud providers are Microsoft and Amazon.

There are many advantages with a public cloud service [24], some of them were briefly mentioned earlier. A major advantage is that one does not have to manage and update the hardware as this is taken care of by the cloud provider. Another advantage is the low startup cost. Acquiring sufficient hardware resources is expensive, especially for small start-up companies, this can be difficult. When utilizing a public cloud provider, it is possible to dynamically scale how much computation resources that are being used, which cannot be done in the same way with a private data center.

2.4.3 Private Cloud

A private cloud implies that one owns and manages one's own servers instead of letting a cloud provider do it. It is possible to purchase cloud stack software from cloud providers to put on top of one's own servers like Azure Stack [25], and it is also possible to use open source cloud stacks such as CloudStack [26]. A software stack can be defined as [27]: "*A software stack is a group of programs that work in tandem to produce a result or achieve a common goal. Software stack also refers to any set of applications that works in a specific and defined order toward a common goal, or any group of utilities or routine applications that work as a set. Installable files, software definitions of products and patches can be included in a software stack.*" A cloud stack refers to all the software components that is needed to build a cloud service.

Although utilizing a cloud stack on one's own servers seems good, it also appears that most of the advantages of the cloud disappears when utilizing a private cloud. It is true that one does limit a lot of the scaling capabilities one gets with a public cloud service, and one has to deal with the acquisition and management of the servers. However, in the long term it might be cheaper to own one's own hardware, granted that the computation power needed is somewhat stable or at least somewhat predictable. The main reason however, for why companies chose a private cloud over the public cloud is security and control [28].

2.4.4 Hybrid Cloud

As the name suggests, hybrid cloud is a combination of both private and public cloud [29]. The hybrid cloud is an important thing to be aware of, as it enables a company to gain all the benefits of the cloud. Highly sensitive information can be kept in-house, while other less sensitive systems and information can be kept in the public cloud to allow for maximal scalability and availability.

2.4.5 The Different Service Levels

Cloud computing is a service that can be provided in different ways and on different levels. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) are the expressions most commonly used. Although there is no clear academic definition to separate the different levels, and different vendors have different definitions [24], it is still valuable to get an idea of the different terms in order to characterize services and understand what goes into them. One of the better explanations of the terms is provided in [30] and is summarized here.

IaaS is the lowest level of service a cloud provider can offer. As the name implies, infrastructure as a service means that the infrastructure is provided by the cloud provider. Here the customer has the ability to rent for example CPU power and storage space, without having to maintain or purchase the physical hardware. Usually the infrastructure is provided in the form of a virtualization platform, where the users have the possibility of creating virtual machines with configurable amounts of computational power, RAM, and storage. This is a service typically used for things like hosting web sites and backing up data. IaaS is mainly used by businesses, but it can also be used by individuals.

PaaS is a type of service that in addition to providing the hardware, also provides an application stack. In software programming, the developer often has to spend a lot of time on handling caching, asynchronous messaging and so on [30]. PaaS includes these kinds of services so that the time can be spent on developing the main functionality. Obviously, relying on such a platform limits the possibilities for optimization, as the developer can only use the application stack that the platform provides. So, optimizing things such as memory access is difficult and maybe even impossible. Often, the platform user cannot decide how much resources that are allocated to the application neither, but as the cloud computing and PaaS industry matures, the development platforms becomes more transparent for the users, and they begin to support a larger variety of programming languages. Many companies use PaaS, for both business analysis and development, but it is also used by individuals.

SaaS is the highest level of service a cloud provider can offer. Here fully developed software programs can be utilized without having to update the software, and with just a minimal amount of configuration. The software will usually be available to the user through a web browser. The user pays for the program typically through a monthly subscription plan. A typical example of SaaS could be Microsoft Office 365, where one can access the newest versions of Microsoft Office without having to install the program onto your own PC. Other programs that are typically offered as SaaS are accounting services for enterprises, and other things that take care of non-core functionality in a business. SaaS cannot be used in a private cloud environment, as the cloud provider will have to for example update the software and would not be able to access it in a private cloud [31].

2.4.6 Security

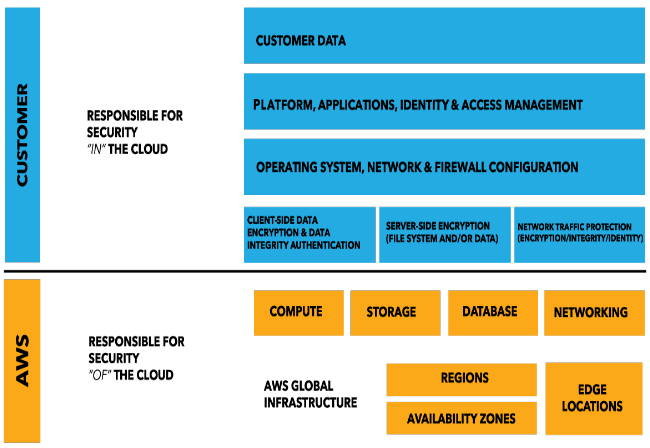
Security of data has previously been mentioned as one of the major challenges with the cloud. Not being careful when utilizing the cloud can have severe consequences, for example in 2014, when the company Code Spaces was forced to give up after a security breach on their AWS account [32].

Gartner has pointed out some of the things that a potential cloud user should investigate, and be aware of, before choosing a cloud provider [33]. It is obviously important to be aware of how the cloud provider protects your data and separates it from its other customers. To know that the cloud provider has experienced encryption specialists, and that they use state of the art encryption methods is something one should look into beforehand. Cloud security really have two sides to it though, the first one is as previously stated, how well they protect your data from others, the other one is how safe is your data from being lost entirely. For example, in the event of your cloud provider going bankrupt, or if an entire data center is damaged beyond repair. One should be aware of the cloud providers plans for your data if it goes broke, and that the company has recovery plans for your data in case of disaster.

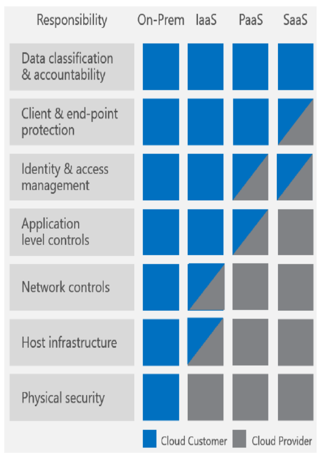
The main security issue, that potential cloud users are the most afraid of, is probably cyber-attacks. Obviously, cyber-attacks are not exclusive to cloud computing, but there are several security considerations and threats that are introduced, or applies to a much higher extent, with cloud computing. The six SOs (Security Objectives) confidentiality, integrity, availability, multi-trust, auditability, and usability are defined in [34].

One example of how one of these SOs, integrity, can be compromised, is with a man-in-the-middle-attack, which can be defined as: "*An adversarial computer between two computers pretending to one to be the other*" [35]. An attack like this allows a third party to snoop on data being sent between two parties, for example, a cloud server and an office desktop.

A big misconception regarding cloud computing is that it is the cloud providers fault every time there is a security breach. In most cases, cloud accounts are hacked because of bad security practices by the user [36], and many cloud users are not entirely aware of how to protect themselves in the cloud [37]. Cloud security is really a shared effort between the cloud provider and the user.


CUSTOMER
RESPONSIBLE FOR SECURITY "IN" THE CLOUD
CUSTOMER DATA
PLATFORM, APPLICATIONS, IDENTITY & ACCESS MANAGEMENT
OPERATING SYSTEM, NETWORK & FIREWALL CONFIGURATION
CLIENT-SIDE DATA ENCRYPTION & DATA INTEGRITY AUTHENTICATION
SERVER-SIDE ENCRYPTION (FILE SYSTEM AND/OR DATA)
NETWORK TRAFFIC PROTECTION (ENCRYPTION/INTEGRITY/IDENTITY)
AWS
RESPONSIBLE FOR SECURITY "OF" THE CLOUD
COMPUTE
STORAGE
DATABASE
NETWORKING
AWS GLOBAL INFRASTRUCTURE
REGIONS
AVAILABILITY ZONES
EDGE LOCATIONS

(A) Overview of Amazon's shared responsibility model [38]



Responsibility	On-Prem	IaaS	PaaS	SaaS
Data classification & accountability	🟦	🟦	🟦	🟦
Client & end-point protection	🟦	🟦	🟦	◩
Identity & access management	🟦	🟦	◩	◩
Application level controls	🟦	🟦	◩	⬛
Network controls	🟦	◩	⬛	⬛
Host infrastructure	🟦	◩	⬛	⬛
Physical security	🟦	⬛	⬛	⬛

🟦 Cloud Customer ⬛ Cloud Provider

(B) Azure's responsibility distribution for the different service levels [39]

Figure 2.3: Responsibility models for cloud security

In Figure 2.3a Amazon's shared responsibility model is shown. This figure states what aspects of security that they are responsible for, and what the customer is responsible for. Figure 2.3b gives an overview on how Azure distribute the security responsibility between them and the customer. Here one can also see how the different service types have different responsibility distributions.

From both parts of Figure 2.3, it is clear that the cloud provider has the responsibility of assuring the physical security of the cloud, i.e. the data centers. This is an aspect of security that may be better with the cloud, compared to private data centers. Cloud providers take great care in making sure that their data centers are secure. The buildings are hard to get into, and if someone were to get in, it would be much harder to extract any useful data. Even though cloud data centers may seem like more desirable targets because of the huge collection of data, they are much easier to protect in a cost-effective manner [34]. From these figures, it is also clear that much of the security is the responsibility of the users, especially with IaaS and PaaS.

2.4.7 VM- and Container Placement

Power consumption is one of the largest costs of operating a cloud data center, at about 42% of the total cost [40]. Hence, it is important for cloud providers to optimize their data centers for power efficiency. As stated previously, cloud providers utilize VMs and containers to offer computing resources to their customers, therefore, it is important to optimize the placement of VMs and containers in a data center. That is, how to optimally distribute the virtual computational resources over the physical hardware available in the data center. Furthermore, it is important for customers of cloud providers to have power and resource efficient solutions, as utilizing inefficient solutions can increase the infrastructure costs significantly. If a company wishes to make a solution available to multiple users, it is important that for each added user, the amount of resources needed to facilitate the new user is as low as possible. Otherwise, it will be difficult to scale a solution to multiple users.

There has been done a lot of research on this topic. Equation 2.1 [41] shows an objective function that can be used to minimize the overall cost of leasing a certain number of VMs and deploying a certain number of containers onto those VMs. The equation is used with the IBM CPLEX Optimizer [42], in order to decide which container c_d should be deployed onto which VM k_v, this is done via the decision variable $x_{(c_d,k_v,t)}$ which is set by the optimizer.

$$\min \Bigg[\sum_{v\in V} c_v \cdot \gamma_{(v,t)} + \sum_{d\in D} \sum_{c_d\in C_d} \sum_{v\in V} \sum_{k_v\in K_v} \big((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \varDelta_d)\big) + \sum_{v\in V} \sum_{k_v\in K_v} \omega_f^R \cdot f_{(R,k_v,t)} + \sum_{d\in D} \sum_{c_d\in C_d} \sum_{v\in V} \sum_{k_v\in K_v} \big(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)}\big) \Bigg] \tag{2.1}$$

The model takes a set of different VM types ($V = \{1, ..., v^{\#}\}$) and container types ($D = \{1, ..., d^{\#}\}$) as input. v and d correspond to a different type of VM and container respectively, while k_v and c_d refers to a specific instance of a VM or container respectively.

The objective function consists of four terms. In the first term

$$\sum_{v\in V} c_v \cdot \gamma_{(v,t)}$$

the overall VM leasing cost is computed. $\gamma_{(v,t)}$ is the number of VM instances of type v with cost c_v and they are leased at a time t. The second term

$$\sum_{d\in D} \sum_{c_d\in C_d} \sum_{v\in V} \sum_{k_v\in K_v} \big((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \varDelta_d)\big) .$$

finds the total time needed to deploy a container ($\varDelta_d$) on a VM (k_v). This is important to take into account, because if a specific container type c_d is deployed on a VM for the first time, the container data needs to be downloaded from the container registry to the VM. However, once the data has been downloaded, it is cached (in the memory) of the VM throughout its lifespan. Therefore, if a container of the same type as before is needed again, this VM will not have to download the data again, hence it will save some time. So, it is clearly beneficial to utilize the same VM for the same type of container. If this is the case, then $z_{(d,k_v,t)}$=1 which makes the product inside the sum 0. In term three

$$\sum_{v\in V} \sum_{k_v\in K_v} \omega_f^R \cdot f_{(R,k_v,t)}$$

the amount of free resources ($f_{(R,k_v,t)}$) are summed up. This term ensures that available resources are used instead of just leasing new VM instances, provided that there are enough resources available in the VM to create another container. ω_f^R ensures that this term is weighted correctly compared to the other terms. The fourth term

$$\sum_{d\in D}\sum_{c_d\in C_d}\sum_{v\in V}\sum_{k_v\in K_v}(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)})$$

sums up all of the deployed containers at a time t. This term aims to provide the minimal amount of resources to each container, while still giving each container enough resources to work. ω_s is the weight of the term.

2.5 FPGA Development

FPGA development is a complex task which requires advanced tools. In this section, the steps in FPGA development will be outlined.

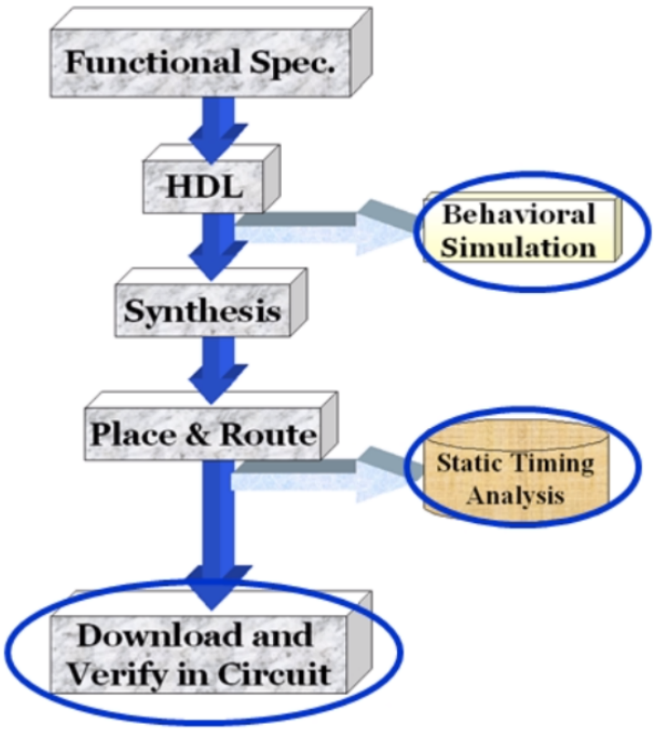


Figure 2.4: Development process for FPGAs [5]

Figure 2.4 shows an overview of the development process. The first step is defining the functional specification of the circuit. After one has defined the circuit specification, one has to write the RTL (Register Transfer Level) code that fulfills the specification in a HDL (Hardware Description Language), typically VHDL or Verilog. There are several ways of verifying the functionality of the HDL code, typically it is done by writing test-benches that provides inputs and checks for correct outputs of a design, and by viewing the waveform of the signals in the circuit to see that they are behaving correctly. The next step in the process is synthesis, which is the process of converting the code into a netlist of FPGA components like for example LUTs [43]. After synthesis is done one can do post-synthesis simulations and get estimates on, for example, power usage and maximal possible clock frequency, and one can see if the results are within the boundaries of the specification. Often, this is not the case, hence one has to go back and make improvements to the HDL code. This is commonly done several times, hence, synthesis is often run multiple times before place-and-route. Once it appears that the circuit meets the demands of the specification, one can move on to the place-and-route, where the component list obtained from the synthesis is mapped to a specific FPGA. After this, one performs static timing analysis, where one verifies that the design actually meets the timing requirements. Then the design is assembled into a binary file, which is downloaded onto the FPGA.

Chapter 3

Previous Work

In this chapter, both commercial products and academic work that this thesis is either related to or based on will be explained. The explanations provided in this chapter are mainly intended to give an overview of the tools that are used in this project and how they are used. Therefore, several details on how the some of the technologies work are omitted.

3.1 Docker

Docker [44] is a software designed to create containers, which were explained in Section 2.2.

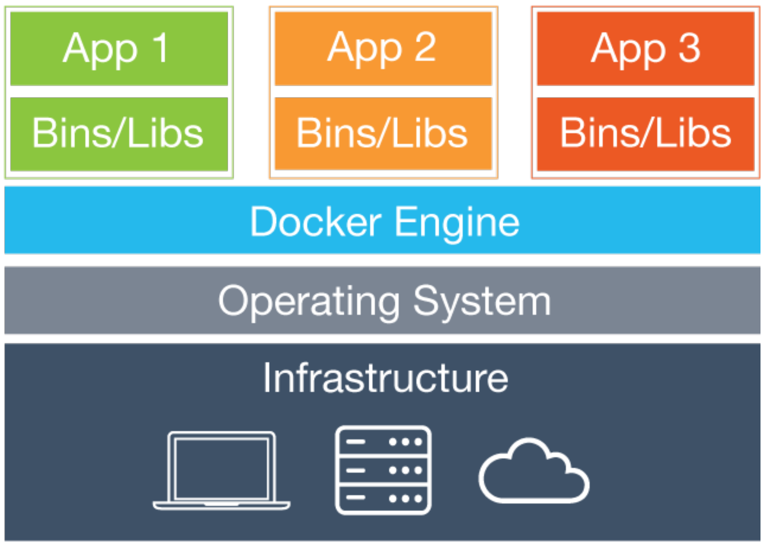


FIGURE 3.1: Docker Engine [6]

Figure 3.1 shows how the Docker engine works as a foundation for the Docker containers. However, it is important to notice that the Docker engine does not work the same way a hypervisor does. A hypervisor manages all calls from a VM to the host OS, while the Docker engine only handles the building and lifetime management of a container. The Docker engine does not handle calls from a container to the host, the removal of this link is one of the reasons why containers are more efficient than VMs.

A Docker container is created from a Docker image. An image is a read-only, executable package, it contains everything needed to run an application [45]. Images are usually created in such a way that everything needed to run an application is prebuilt. Therefore, creating a container instance to run an application is an almost instantaneous process. A Docker image is built up in layers. Each new layer in a Docker image is added on top of the other layers. This means that if one has one or more layers that make up an OS like Ubuntu 14.04, and then in the next layer, installs for example Python, Python will be installed in the Ubuntu environment.

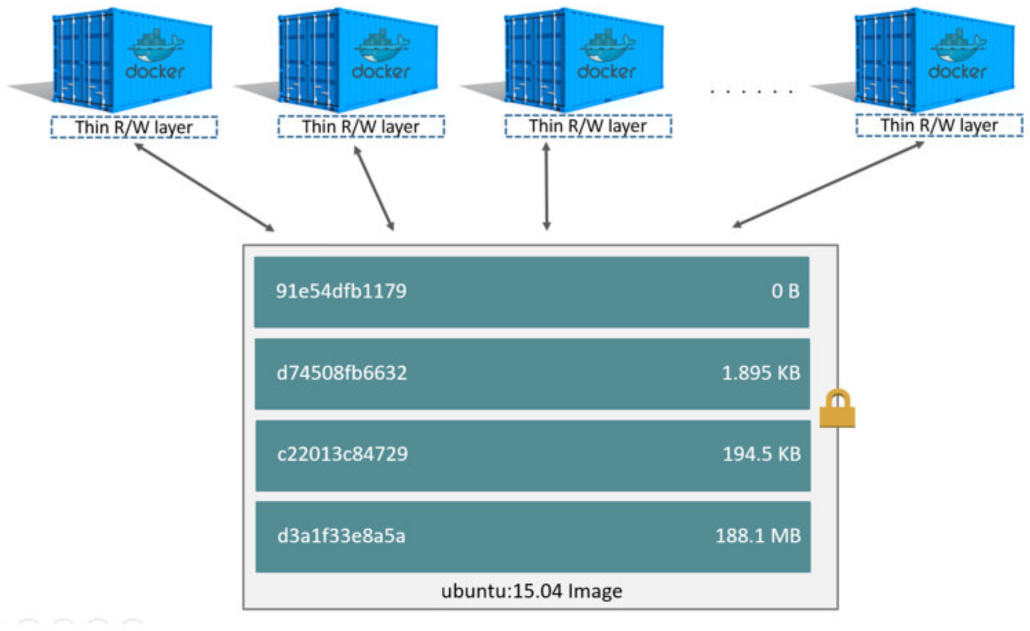


FIGURE 3.2: An overview of how a Docker image is divided into layers and can be used by multiple containers [7]

Figure 3.2 shows how the Ubuntu 15.04 image consist of four layers. It also shows how multiple Docker containers utilize the same image to run simultaneously. As already stated, a Docker image is read-only, a container only consists of a small read/write-layer on top of the Docker image, allowing one to access and use the applications within the

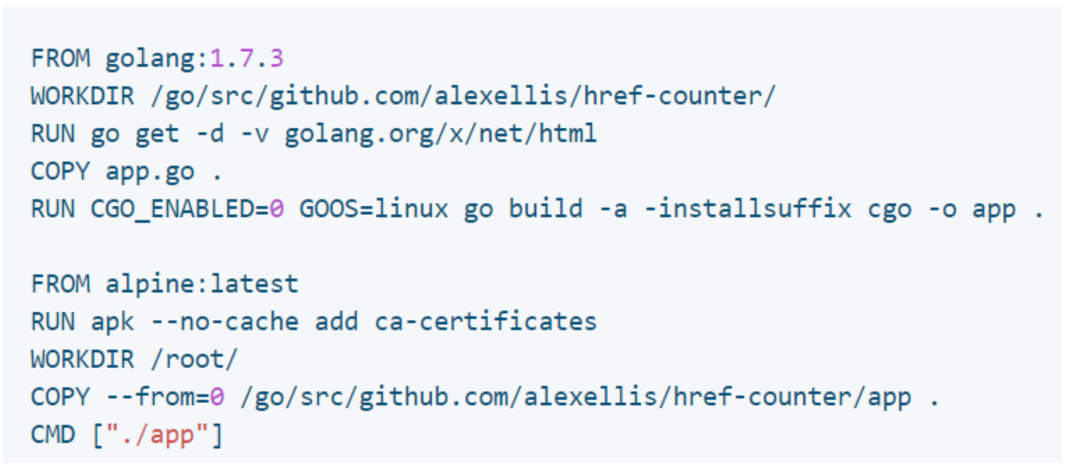
Docker image. All Docker images are stored in the same place and managed by the Docker engine, along with this, the layer-architecture of Docker images also allows for efficient use of disk space. If one, for example, has two different images that both utilize Ubuntu 14.04, then one of the images has Apache web server installed, while the other one only has Python installed, each layer in the images is only stored once This means that the Ubuntu 14.04 image is only stored once, regardless of how many images that depends on it.

To say that only one Docker image is needed once, and that each layer in the images is stored only per machine, is somewhat of a simplification. In actuality, each layer is stored once per Docker engine, of which there is typically only one of per machine. However, to scale Docker containers to multiple users, often multiple Docker engines are needed. Because, even though there can be multiple Docker containers running on top of the same image, when enough containers try to access the same image at the same time, there will be latency. Docker Swarm [46] is a software that is able to do handle this and create multiple Docker engines and copy over the necessary images to each engine.

A Docker image is created from a Dockerfile. A Dockerfile defines the environment inside a container, and hence, how the Docker image should be built [47]. Each line in the Dockerfile adds one or more layers to the Docker image, it always starts with the FROM expression which indicates the basis of a new image. For example, writing the line “FROM ubuntu 14.04” will create an image that is based on the Ubuntu 14.04 image that already exists. It is also possible to use the expression ”FROM scratch” if one wish to create a new image entirely, but this is not very common. The images that already exist are often fetched from a website called docker hub [48], which is a page where companies and individuals can publish Docker images.

There are a handful of expressions used in Dockerfiles [49], in this thesis only a few of them are explained. The FROM expression has already been covered. Another important instruction is RUN. It is the instruction that is used to run any command within the Docker image. Each time the RUN Instruction is used a new layer is created on top of the current image. One more instruction to be aware of is WORKDIR, which changes the working directory within the image. One can say that using the WORKDIR instruction is the Dockerfiles way of using the cd command in a terminal.

Multi-stage build is a technique used in Dockerfiles to create Docker images. As previously mentioned, each new line in a Dockerfile adds a new layer to the Docker image. Hence, all packages and files used in a Dockerfile will be in the final image. Even if they are removed or uninstalled and hence will not be available inside a container once it is running, the image will still not be reduced in size. This may lead to an unnecessary large image, as it will typically be full of tools that were only needed to install the application inside the image but serves no function once the application has been installed. To avoid this problem, one can utilize multi-stage builds [8]. Multi-stage builds are created by using multiple FROM statements in the same Dockerfile. A typical way of doing this, is to make the first image in the file very much like one would do in a single-stage build, with all installation tools included. Then, bellow the first image, create a new image where one only copies the necessary parts from the first image. In Figure 3.3 an example of a multi-stage Dockerfile is shown.



```
FROM golang:1.7.3
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

FIGURE 3.3: An example of multi-stage builds [8]

The figure here is both an example of how a Dockerfile looks in general as well as what multi-stage builds are. On the first line, it is shown that the base image used is the golang:1.7.3. Note that the software used in this Dockerfile example, will not be elaborated on here. The Dockerfile then changes the working directory, runs a command, copies an application file (app.go) and then uses another RUN statement to install it. To minimize the size of the image, in the second build stage, it is used another FROM statement that utilizes the alpine base image instead of the golang image. In the second to last line it copies the folder where the app is installed from the previous image. The "–from=0" flag indicates that this image copies from the image created with the first

FROM statement. Each sub-image in a multistage image is numbered, starting from 0. It is also possible to tag each sub-image with a name to make it easier to see what image one is copying from.

When using Docker, one can use the “docker build” command to create a new image from a Dockerfile. One can either specify a path to the Dockerfile, or the docker build command will automatically detect the Dockerfile if there is one in the current directory. To run a container, one can use the ”docker run” command. To measure the resource usage of containers, Docker has a command ”docker stats” [50]. The command displays real-time resource usage from all running containers. It can also be used along with other software to collect stats over a period of time. The stats that are displayed are CPU usage, memory usage, network I/O and block I/O.

3.2 IDE8

IDE8 is a cloud-based development environment. It makes it easy to develop products and services based on electronics and mechatronics. It is a relatively new development platform, developed by *Wireless Trondheim*. IDE8 can be defined as a PaaS, as it offers many different development platforms. It is important to notice that IDE8 is still in very early stages of development. Therefore, new features are constantly added, and a lot of functionality within IDE8 is subject to change.

The development environment of a user in IDE8 is called a Workspace. One developer can have multiple Workspaces. Each Workspace is created from one of the predefined stacks found in the IDE8 library. A stack defines what tools and services that will be available to the user within the Workspace. A Stack can be optimized for a product like an Arduino or a Lattice FPGA, an application, e.g. a heart rate sensor, an event, e.g. an IOT hackathon, or a group of people, e.g. students.

The other important part of IDE8 is the IDE8 board agent. The agent connects a device, for example a development board or an oscilloscope, to a user’s Workspace. This agent can be downloaded to a local PC allowing the user to connect his or her own device to his or her own IDE8 Workspace. *Wireless Trondheim* also has several devices already connected to the IDE8 site, that are free to use for anyone.

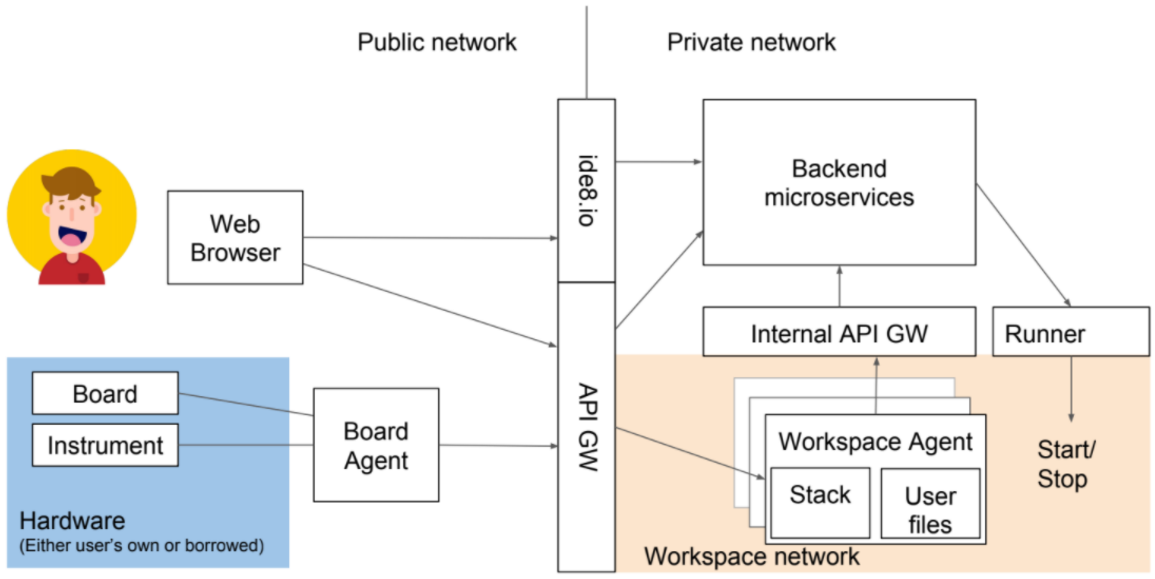


Figure 3.4: Overview IDE8 architecture [9]

Figure 3.4 shows an overview of the IDE8 architecture. On the left side of the figure we see that the user accesses IDE8 through a web browser, and it is also shown that the physical boards and instruments are connected to the board agent over the public network. The connected devices can either be the user's own device connected directly to the user's PC, or it can be one of the aforementioned devices that IDE8 already has connected to the API GW (Application Programmable Interface GateWay). On the right side, we see the internal workings and structure of IDE8. The *Backend microservices* box is the backend infrastructure that IDE8 runs on, which is provided by the IaaS provider Digital Ocean [51]. On the bottom right, it is shown that the Workspace Agent is connected to the infrastructure, user, and Board Agent through the two different APIs, it is also shown that the Workspace contains both the Stack and the user's own files. There is also a Runner connected to the backend which controls the lifespan of the containers within IDE8.

A stack in IDE8 is a build environment inside a Docker image. IDE8 default Workspace stack uses a Debian stretch image, a Linux based OS, as its base image, along with some additional tools like git and make [9]. A custom stack like an Arduino stack runs on top of the default Workspace stack. The definition of a stack must be put in a file named stack.yaml located in the /ide8/ directory of the base image. This is done to describe the commands that can be run in the stack and what examples that are available.

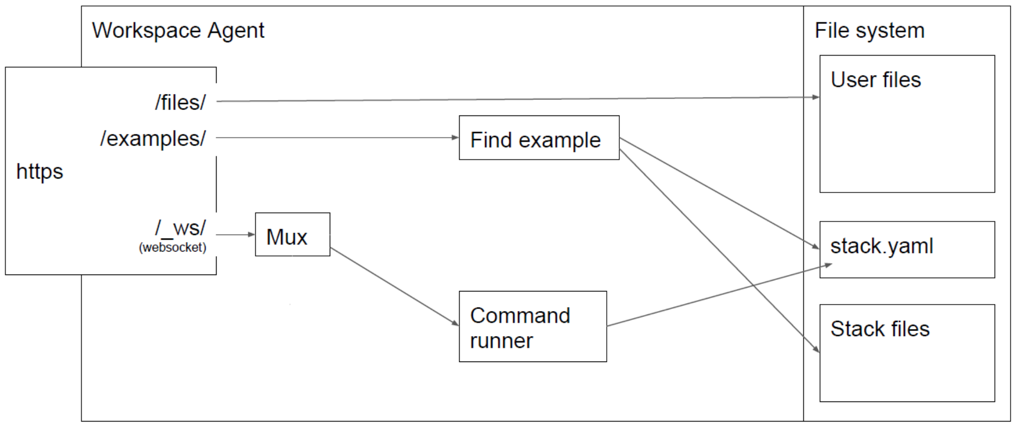


FIGURE 3.5: Overview of Workspace Agent components [9]

Figure 3.5 shows some of the details of the IDE8 Workspace agent. The Workspace agent is the name of the system that creates a Workspace. One can see that the user is exposed to two different set of files in the web browser. The directory called "files" are the user's own files, typically the source code for a specific project. The other folder, "examples", are specified in the stack.yaml file and then fetched from the Stack files. These examples can be imported to the user files if the user wishes to try out the examples. Each stack comes with a different set of examples. The command runner is accessed from the webSocket [52], which is a communication protocol for two-way communication over the web, and as mentioned previously, the commands must be defined in stack.yaml. The multiplexer (Mux) in the figure, is there so that the webSocket can perform other tasks as well, they are however, omitted in this figure.

3.3 AWS EC2 F1 and Developer AMI

In 2016 Amazon Web Services (AWS) started a collaboration with Xilinx, one of the two major FPGA providers [53]. Xilinx FPGAs have been integrated into some of AWS's data centers. The instances have been made available for AWS users, allowing them to utilize FPGAs to accelerate computations. These FPGA instances are named EC2 F1 (Elastic Compute Cloud F1). The F1 instances consist of one or more FPGAs and a virtual machine. Along with these instances, AWS has created the FPGA developer AMI (Amazon Machine Image), which is a software package that can be used with any of

AWS's VM instances, and provides Xilinx's Vivado development tool, along with some additional features.

EC2 is the general name of AWS virtual machine instances, of which there are many [54]. Each instance type has a given set of specifications: RAM, CPU cores, storage and also several other attributes. In the case of F1 instances, they come in two different types with different attributes which are listed in table 3.1

TABLE 3.1: Specifications for F1 instances [14]

Instance	FPGAs	DDR-4 (GiB)	vCPUs	Memory (GiB)	Storage (GiB)	Bandwidth
f1.2xlarge	1	4x16	8	122	1x480	10 Gbps Peak
f1.16large	8	32x16	64	976	4x960	30 Gbps

Notice that GiB is used here and not GB. GiB, or Gibibyte, is per definition 2^{30} bytes while a GB, or Gigabyte, is part of the SI standard, with the value of 10^9 bytes [55].

It is shown that the smaller one of the F1 instances only comes with one FPGA, while the large one comes with eight. The eight FPGAs on the larger instance are also interconnected and can communicate directly with each other. The FPGAs on the F1 instances have the following specs [53]:

- Xilinx UltraScale Plus 16nm FPGA
- 64 GiB DDR4 ECC-protected memory
- dedicated PCIe x16 connection
- 2.5 million logic elements
- 6800 Digital Signal Processing engines
- Virtual JTAG interface for debugging

The table also shows that each FPGA has 4 times 16 GiB of DDR4 memory attached, which sums up to 64 GiB per FPGA. This allows the FPGAs to store quite large amounts of data without having to communicate with the processor of the VM. Each VM of the smaller and the larger instances has 8 and 64 virtual CPU cores respectively, also 122 and 976 GiB of Memory and 480 and 3840 GiB of Storage. The large instance also has a network bandwidth of 30 Gbps while the small one only has a peak bandwidth of 10 Gbps.

The F1 instances are intended to be used for running applications that utilize FPGAs for accelerated computing. They are not however, normally used for developing the FPGA design. As seen earlier, the F1 instances comes with rather heavy specs and are hence very expensive to run. The larger one of the F1 instances has, as of May 2018, an hourly renting price of $13.2 [56], making it one of the most expensive out of all EC2 instances offered by AWS. To develop FPGA designs, one typically uses a smaller instance type, with specs closer to that of a normal desktop computer, for example the t2 instance which is priced in the range from $0.006 to $0.37, depending on the amount of memory and CPU cores that are needed. The t2 instance does not have an FPGA connected to it, but for most of the design process this is not needed. It is typically enough to utilize simulation and debugging tools.

When developing an FPGA design in AWS, one will utilize the AWS FPGA developer AMI [57]. AMI is short for Amazon Machine Image, which is an already configured software package that can be run on a VM. It is similar to Docker images for containers, but instead of describing how a container should be, it defines how a VM should run. It contains the OS the VM should run, and the software packages which will be installed on the machine. The FPGA AMI runs CentOS 7.4 which is a Linux/Unix based operating system, and it comes with the Xilinx FPGA developer tool Vivado and the AWS-FPGA HDK (Hardware Development Kit). The HDK and SDK (Software Development Kit) is available on GitHub [58].

The HDK is a set off tools that is designed to make it easier to integrate and deploy the FPGA design on F1 instances. The HDK contains the tools needed to create an AFI (Amazon FPGA Image) from an FPGA design. An AFI is similar to an AMI, but instead of containing a setup for a VM, it contains the design that is to be implemented on an Amazon F1 FPGA. An AFI can also utilize more than one FPGA, as previously stated the largest F1 instance has eight FPGAs available, and one can use the same AFI to deploy a design that utilizes more than just one of the FPGAs. The HDK also contains the FPGA shell interface [10], which is the logic interface used to communicate between the FPGAs and the peripherals, like the DDR4 memory and the PCIe.

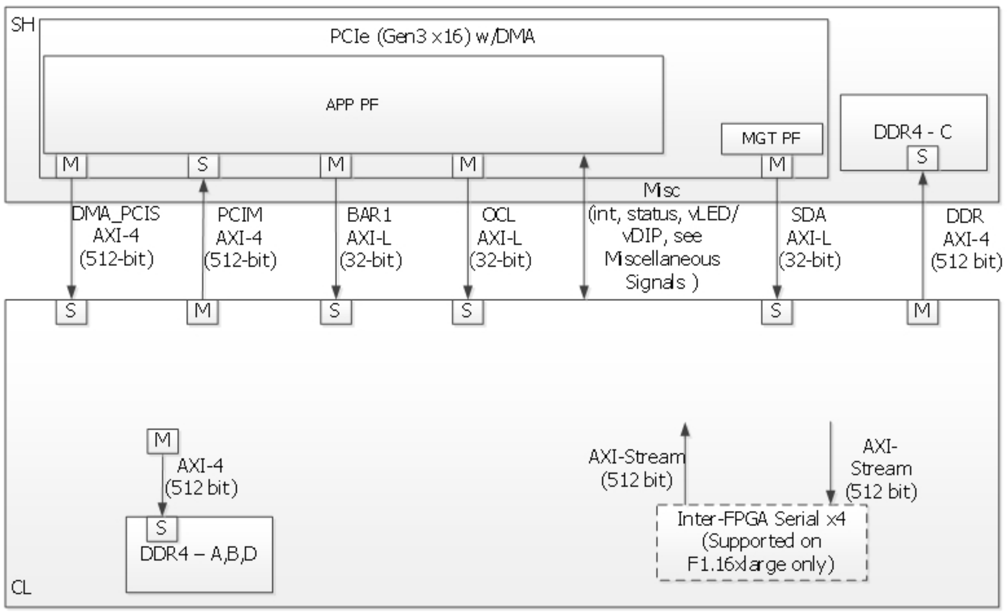


FIGURE 3.6: Summary of the AWS FPGA shell interface [10]

Figure 3.6 contains a detailed overview of the shell interface. The figure will not be explained in detail here, but some of the main points will be explained. The figure shows two large boxes, the top box is labeled SH for shell and the bottom one is labeled CL for custom logic, which is the actual FPGA logic created by the FPGA developer. The main part of the Shell is the PCIe interface, but the figure also shows that there is an interface between the FPGA and the DDR4 memory. In addition to this, if one is to use the f1.16xlarge instance with eight FPGAs, the shell also includes an interface for inter-FPGA communication.

The SDK is found on the same git repository as the HDK. The SDK is made for managing the AFIs, not create them.

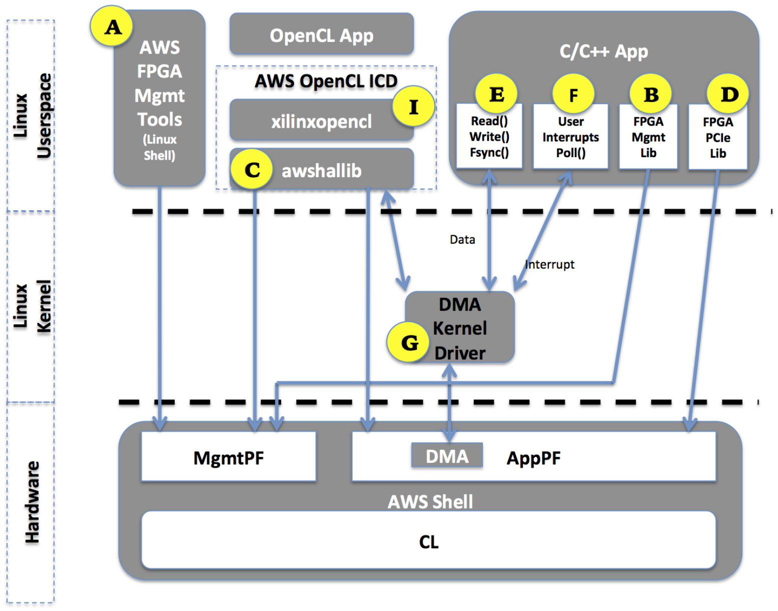


FIGURE 3.7: Summary of the AWS FPGA software [11]

Figure 3.7, provides an overview of how the SDK is used to manage and work with the FPGAs. At the top of the figure is the Linux Userspace, which is the management tools and runtime communication tools that the developer uses to access the FPGA. A, B and C on the figure makes up the FPGA management interface. Point A on the figure is Linux shell commands which can be used to do operations like loading and removing AFIs and also activate the virtual JTAG to do on-chip debugging. Point B is a C-library which is to be compiled with a developer's C/C++ application. Point C is a library with the OpenCL runtime library pre-integrated. The points D, E, F and I are all different libraries used for different types off runtime communication with the FPGA. D, E and F are for C/C++ applications, while I is for Open CL. G is the DMA (Direct Memory Access) driver that is needed for E and F to work.

The development process can be summarized in four steps [14]. The first step is to launch an AWS virtual machine with the FPGA developer AMI and do the necessary configurations for the HDK. The second step is the actual development of the design, where one utilizes the Vivado software to write either Verilog or VHDL code. One can also utilize HLS (High-Level Synthesis) with OpenCL [59]. These tools should be used along with the HDKs FPGA shell to create the full logic design. After the design is complete, and synthesizing and simulations are done, the third step is to do the place-and-route which generates a design checkpoint file (DCP). The DCP contains the complete design which again can be turned into an AFI, which is encrypted. Then in step four, the AFI can be loaded onto a F1 instance. The AFI can then be managed with the AFI management tools in the SDK. The FPGA developer AMI can also be used for normal FPGA development, not aimed at the F1 instances.

The AWS marketplace is a service where AWS users can offer their custom made AMIs and AFIs to other AWS users. Also, AWS offers their official AMIs like the FPGA developer AMI through the marketplace. There are several businesses that now uses the AWS marketplace to distribute their solutions. One example is the company Mipsology, which were one of the first to offer a solution which utilized FPGAs. The application they offer uses an FPGA for inference of neural networks, which is used for image classification [60]. However, the solution is offered as an AMI not an AFI, this is because the solution contains the entire setup for the VM, so the AFI is included in the AMI.

3.4 FPGA Infrastructure Solution

In the paper *Using Clouds for FPGA Development - A Commercial Perspective* [12], an alternative solution to AWS's FPGA developer AMI and F1 instances was presented. They also had a look at current cloud-based FPGA products, including the AWS F1 instances. They examined, on a theoretical level, the possibilities of having the cloud providers invest into an FPGA farm and deliver FPGAs as an IaaS.

This solution is mainly intended for companies that does not have FPGA development as one of their primary focus areas but needs it as a part of a bigger system, for example an encryption accelerator on a System on Chip (SoC) design. These companies may not have much experience with FPGA testing, so acquiring hardware and setting up a test environment can be cumbersome. Furthermore, they may need several FPGAs to test different things at once. This solution will enable faster and less expensive testing, as the user would be able to test on multiple FPGAs at once, without having to acquire or deal with additional hardware. Their paper outlines further details on this solution and provides an estimate for how long it will take to develop this service.

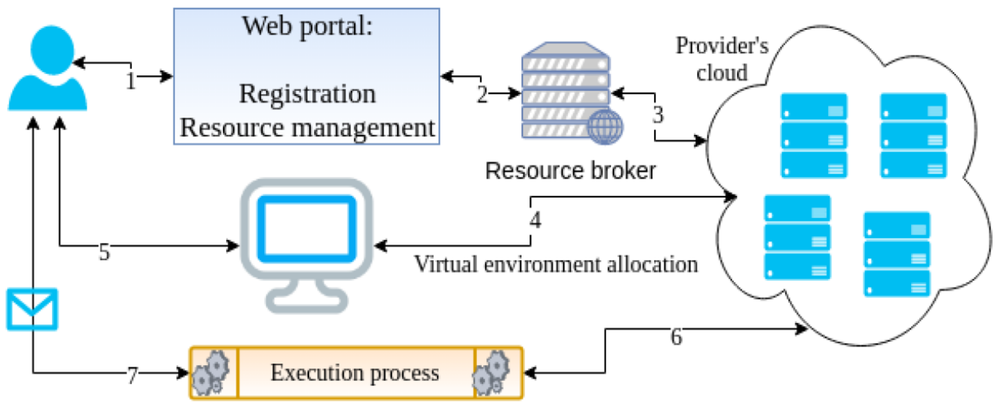


FIGURE 3.8: Overview of solution processes [12]

Figure 3.8 shows a schematic overview of the solution. The solution works quite similar to other cloud solutions. The registration and resource management are done through a web portal (1), and the interaction between the user and the cloud resources is done through an interactive virtual environment (4)(5). This virtual desktop which is displayed to the user will contain necessary tools like an IDE to develop the project, much like how the AWS FPGA developer AMI works. After the user is done with the project, or at least wishes to test it, the project can be launched for synthesis, place-and-route, and bit stream generation inside the cloud (6)(7). This process will be done in the background, allowing the user to continue working on other projects or other parts of a design while the project is being built. As mentioned earlier, the provider's cloud also contains FPGAs, allowing the design to be tested on physical hardware.

3.5 Project IceStorm

Project IceStorm is a fully open source design flow, from Verilog to bitstream, for the Lattice iCE40 FPGA [13]. It consists of three different parts, the synthesis tool Yosys, the place-and-route tool Arachne-pnr and the IceStorm toolkit.

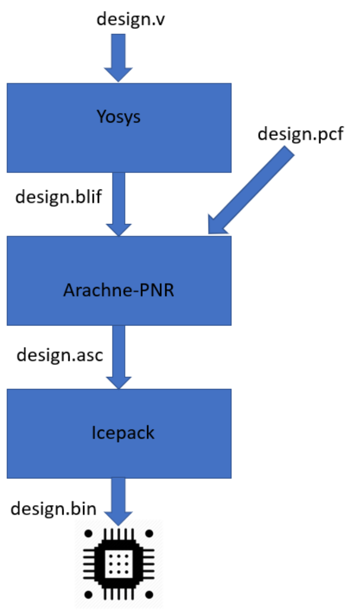


FIGURE 3.9: Overview of the Project IceStorm design flow

Figure 3.9 shows a simple overview of how the Project IceStorm flow works. It functions by utilizing the synthesis tool Yosys to take in the verilog file (design.v) and producing a design.blif file. This file together with a pin constraint file (design.pcf) is processed by the place-and-route tool Arachne-pnr. This produces an ASCII file (design.asc) which the IceStorm tool icepack converts to a design.bin bitstream which is used to program the FPGA.

3.5.1 Yosys

Yosys is the synthesis tool that is used in the IceStorm Project. It was developed around 2013-2014, prior to the development of the rest of the IceStorm project, but by the same project lead, Clifford Wolf [61]. The synthesis tool can take in verilog files and outputs a BLIF file (Berkeley Logic Interchange Format). The BLIF file format, is used to describe logic-level hierarchical circuits in a textual form [62]. This tool is not specifically designed for the Lattice iCE40 FPGA, as both the verilog input code and the BLIF output representation are hardware independent, but since the Arachne-pnr tool takes a BLIF file as an input, the tool is simple to integrate with the rest of the Project IceStorm design flow.

3.5.2 Arachne-PNR

Arachne-pnr is the tool that takes care of the place-and-route step in the FPGA compilation process [63]. It has several dependencies to the IceStorm software, and hence it also targets the Lattice iCE40 FPGAs. The tool takes in a BLIF file and outputs a textual representation of a bitstream. It uses a simulated annealing-based algorithm to execute the place-and-route operation.

3.5.3 IceStorm

The IceStorm tool contains several different tools. The two main tools in the design flow are the icepack and the iceprog tools. The icepack program converts ASCII files into .bin files, which is the raw bitsream format that is used to program the FPGA. The iceprog tool is a small driver program for the programmer that is used on the iCEstick and HX8K development boards.

One other tools that is included is the IceUnpack program which converts a file from .bin to .asc, which can be useful since all the other Ice-tools utilize the ASCII file format.

3.6 Other Open Source Tools

There are several other open source tools available for FPGA development. The tool, Odin II [64], is another synthesis tool for for Verilog code. It produces a BLIF netlist as an output, which is the same format as Yosys. Icarus Verilog [65] is another synthesis tool for Verilog. this tool can generate netlists in several formats. Another place-and-route tool that is available is VPR [66].

Chapter 4

FPGA Tool Chain Solution

The main objective of this thesis work is to build a working prototype of an FPGA tool chain that can be run in the cloud (IDE8). In this section, there will first be explained the different parts of the stack that are needed, along with some optional components that could be interesting to add in the future. Then, the main functionality of the solution will be explained. The FPGA tool chain was implemented with Docker containers in three different ways. In Section 4.3 the implementations will be explained, and in 4.4, the different implementations of the solution will be compared. Note that, throughout the rest of this thesis, it will be referred to both the FPGA tool chain and FPGA stack. While the tool chain mainly referrers to the actual FPGA tools, and the stack referrers to the FPGA tools wrapped in Docker containers, the two terms can be used interchangeably.

4.1 Stack Components

In Section 3.5, the open source tool chain, Project IceStorm, was explained. This is the tool chain that will be used in this implementation.

There are several reasons why Project IceStorm was considered to be a good option. The main reason is that it is an open source solution, without any license issues. Deploying for example Vivado or Lattice Diamond in a cloud solution would require permission from the software providers, which could be a time consuming and cumbersome process, but this will be discussed further in Chapter 6. There are multiple open source options in existence, as stated in Section 3.6, but Project IceStorm was still deemed the best choice, as it takes care of the entire flow from Verilog to bitstream. Even though it is fully possible to exchange for example the Yosys synthesis tool with Icarus Verilog, it would require some additional work. Without any substantial indication that swapping out any of the tools would lead to a performance improvement and because of the limited time frame of this project, it was not deemed necessary to try any other components.

As explained in Section 3.5, Project IceStorm consist of three different software components, Yosys for synthesis, Arachne-pnr for place-and-route and IceStorm for assembling and various other minor functions. These will be the main components of the stack, and the only components used in this thesis. The details on how the stack is implemented is described in Section 4.3.

A fully functioning FPGA development environment however, requires several other components than the ones provided with Project IceStorm. The IDE is of course provided in the Workspace in IDE8, but there are several other tools as shown in Section 2.5. There is a need for tools for debugging, simulation, and visualization. By visualization it is referred to functionality such as schematic view and layout of the design and waveform viewing. Furthermore, another feature that would be interesting to add is HLS (High-Level Synthesis), this will be discussed further in Chapter 6.

4.2 Development Environment

This section will explain the development environment that was used to implement the FPGA stack, as all the development was done locally before being tested with IDE8. How testing and integration with IDE8 was done, is explained in Chapter 5.

To build and test the FPGA stack, a Linux based OS was used since Docker containers are mainly Linux based, even though support for containers based on other OSs does exist. Furthermore, IDE8 is utilizing Linux based containers. To create an isolated development- and testing environment, a Virtual Machine (VM) was used. We used an Ubuntu 16.04 64-bit OS on the VM [67]. To generate and manage the VM the VirtualBox software [68] was used.

The FPGA stack was developed utilizing Docker containers. As explained in Section 3.1, Docker container are running instances of Docker images, and Docker images are read-only executable packages that fully define the environment of the container. The image contains all the files needed to run the container and the application within the container, and a Docker image is defined and built from a Dockerfile. The Docker community edition (CE) software was installed on the VM [69], so that the Docker images could be built, and the containers could run. The Dockerfiles are typically not very large in terms of lines of code, so there was no need for any type of special IDE with debugging features to write the code. However, to build an image from a Dockerfile can take some time, since everything that is put into a Docker image has to be installed. A nice feature in Docker however, is that the different layers of the image are cached, so that if one has to make small changes to an image, one might not have to build the entire image from scratch again.

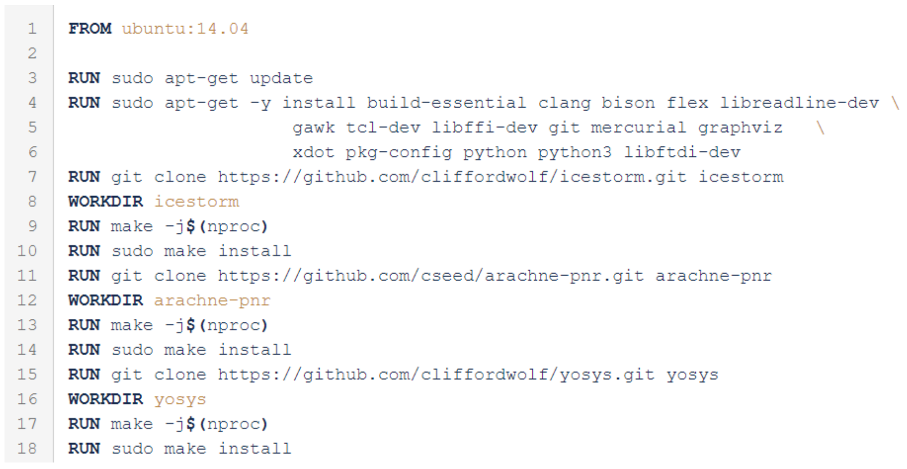
4.3 FPGA Tool Chain

The implementations of the FPGA tool chain are based on Docker containers to make the solutions scalable and functional in the cloud. One does not have to utilize containers to run software in the cloud, but it provides a good way of making sure that the software behaves correctly in different environments as explained in Section 2.2 and 3.1.

The Project IceStorm software has here been installed to run in containers in three different configurations, one monolithic approach and two different microservice approaches.

4.3.1 Monolithic Approach

The first architecture that was implemented was the monolithic approach. In this approach, the entire tool chain, from synthesis to bitfile generation, was put in the same image. When all the tools are put in the same environment, the process of building the Docker image is quite similar to installing the tool on a normal desktop.



```
1  FROM ubuntu:14.04
2
3  RUN sudo apt-get update
4  RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
5                      gawk tcl-dev libffi-dev git mercurial graphviz   \
6                      xdot pkg-config python python3 libftdi-dev
7  RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
8  WORKDIR icestorm
9  RUN make -j$(nproc)
10 RUN sudo make install
11 RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
12 WORKDIR arachne-pnr
13 RUN make -j$(nproc)
14 RUN sudo make install
15 RUN git clone https://github.com/cliffordwolf/yosys.git yosys
16 WORKDIR yosys
17 RUN make -j$(nproc)
18 RUN sudo make install
```

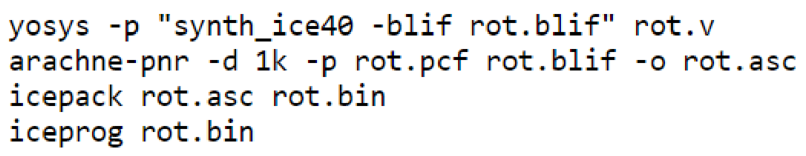
FIGURE 4.1: Dockerfile for building the IceStorm project in a Docker container (monolithic approach)

Figure 4.1 shows the Dockerfile that builds the Docker image containing the tool chain. The Dockerfile defines all the steps necessary to create the image with the desired attributes.

The first line in the Dockerfile defines the base-image used to build the environment. As explained in Section 3.1, all Dockerfiles must begin with a FROM statement. Because Project IceStorm is designed to be installed on Ubuntu 14.04, this was clearly the simplest base-image to use, as it would be able to support all the features of Project IceStorm and hence, reduce the risk of bugs appearing when both creating the Docker image and running the FPGA software in the container.That the Ubuntu image is used, might seem as if the advantages of running a container is removed, as it now appears that an entire OS has been added to the container, making it very similar to a VM. However, this Ubuntu image is not the same as a full OS, this image only contains the binaries and libraries of Ubuntu 14.04, hence, making it a lot more lightweight than a full OS. The size of the Ubuntu image is approximately 220 MB, while the full desktop Ubuntu image is approximately 1 GB in size and requires a lot more disk space than that once it is installed. So, an Ubuntu Docker image is not the same as a full version of Ubuntu. The next two RUN statements on line 3 and 4, are commands run within the Ubuntu 14.04 environment. The first line updates the package list, and the second installs all the packages necessary to install and run the FPGA tools. These commands were available from the installation guide on the Project IceStorm website [13]. The only change done on these two commands were that the -y flag was added on the second command so that one did not have to manually agree to install the packages while the

Docker image was being built. The rest of the Dockerfile handles the installation of the FPGA tools. All of these instructions are also from the Project IceStorm website. On line 7, 11 and 15, the git repositories of each of the software components are cloned into the Docker image. On lines 8, 12 and 16, the WORKDIR command is used, as explained in Section 3.1, this command is used instead of the Linux command “cd” to change the directory. Then the tools are installed using Make [70]. Due to the installation order, we can here see that Yosys is installed within the arachne-pnr directory which is again installed within the icestorm directory. This is not necessary, but it also does not cause any problems. Hence, it was deemed unnecessary to change it.

To test that the tools were functioning properly, a simple Verilog example that is included in the Arachne-pnr git repository was used. The example is a simple Verilog script rot.v along with a placement constraint file, rot.pcf, which blinks the LEDs on the Lattice iCE40 iCEStick development board in a rotating manner.



```
yosys -p "synth_ice40 -blif rot.blif" rot.v
arachne-pnr -d 1k -p rot.pcf rot.blif -o rot.asc
icepack rot.asc rot.bin
iceprog rot.bin
```

Figure 4.2: Instructions for running and creating a binary file from the rot.v Verilog example [13]

Figure 4.2 shows the instructions used to execute synthesis, place-and-route, bitfile generation and programming the FPGA. These were the commands used to verify that the containers were working.

4.3.2 Microservice Approach

The second tool chain implementation is the microservice approach. Here, it is attempted to run Project IceStorm by installing and running the different software components in different containers. However, Arachne-pnr depends on several files from the IceStorm tool to be installed so it would be difficult to run these two programs in different containers. Therefore, the design flow will here only be split into two containers instead of three. The first container will contain the Yosys synthesis tool, while the second will contain both Arachne-pnr and IceStorm

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                    gawk tcl-dev libffi-dev git mercurial graphviz   \
                    xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/yosys.git yosys
WORKDIR yosys
RUN make -j$(nproc)
RUN sudo make install
```

FIGURE 4.3: Dockerfile for building the container with the Yosys tool

Figure 4.3 shows the Dockerfile describing the image containing the Yosys tool. To create it, the same approach as in the monolithic case is taken. The same base-image, ubuntu:14.04, is used, and all the same packages are installed. It is not given that all of the packages are needed here, but for simplicity, all the packages were installed. Furthermore, only the Yosys git repository is cloned and installed, contrary to the monolithic case.

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                    gawk tcl-dev libffi-dev git mercurial graphviz   \
                    xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR arachne-pnr
RUN make -j$(nproc)
RUN sudo make install
```

FIGURE 4.4: Dockerfile for building the container with the IceStorm and Arachne-PNR tools

Figure 4.4 shows the Dockerfile used to create the image with both Arachne-pnr and IceStorm installed. The Dockerfile is also built up very similar to the monolithic file. Here the cloning of the Yosys git repository and installation is simply omitted.

In Section 3.1 a technique called multi-stage builds was explained. This technique is used by having multiple ”FROM” statements in the same Dockerfile. Hence, creating multiple images in the same Dockerfile and only keep certain things from each sub-image in the final image.

```
1   FROM ubuntu:14.04 as builder
2   
3   RUN sudo apt-get update
4   RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
5                        gawk tcl-dev libffi-dev git mercurial graphviz   \
6                        xdot pkg-config python python3 libftdi-dev
7   RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
8   WORKDIR icestorm
9   RUN make -j$(nproc)
10  RUN sudo make install
11  WORKDIR /
12  RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
13  WORKDIR /arachne-pnr
14  RUN make -j$(nproc)
15  RUN sudo make install
16  
17  
18  FROM ubuntu:14.04
19  COPY --from=builder /icestorm /icestorm
20  COPY --from=builder /arachne-pnr .
21  COPY --from=builder /usr/local/bin/i* /usr/local/bin/
```

FIGURE 4.5: Dockerfile for building the container with the IceStorm and Arachne-PNR tools, utilizing multi-stage build

Figure 4.5 shows the Dockerfile that creates the image containing Arachne-pnr and IceStorm by utilizing multi-stage builds. We see that the first part of the Dockerfile, from line 1 to 15, is almost identical to the Dockerfile in Figure 4.4, the only difference being line 1, where it was added "as builder". This is just a way of naming the different stages in a multi-stage build, which makes it easier to see what sub-image is used to create the final image. The second build stage starts at line 18, this stage also utilizes the Ubuntu 14.04 base-image. Then the image uses the COPY command in order to copy only the necessary folders from the previous sub-image. Notice the flag "–from=builder" refers to the first sub-image, which was named builder. To get both Arachne-pnr and IceStorm working in the new image, the icestorm and the arachne-pnr directories are copied into the new image. On line 21 the IceStorm binary files that are generated during the installation are copied into the new image as well. Utilizing the full ubuntu 14.04 base-image in the final image might not be necessary, there are other more lightweight Linux based base-images that probably could have been used, hence produced even less overhead, but to test this further was not prioritized in this thesis.

The Yosys installation process is quite complex and generate several files in several different locations, which makes it difficult to find all the necessary files to bring over to the next build stage. Hence, reducing the image size of the Yosys image utilizing multi-stage builds proved to be a difficult task. Therefore, it was not done in this thesis.

For the same reason, the monolithic approach was not improved with multi-stage builds, as it would be difficult to get Yosys to function properly.

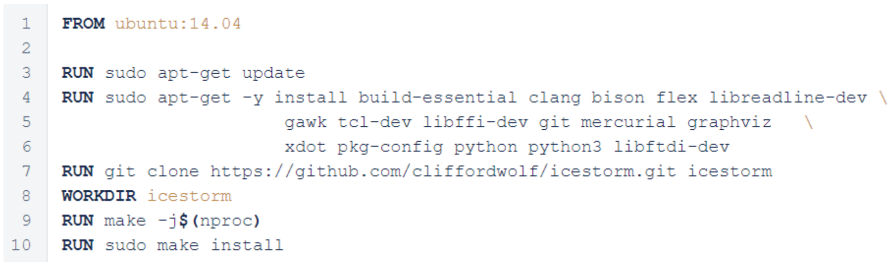
4.3.3 Microservice Approach 2

In the previous approach, it was stated that Arachne-pnr relies on files from IceStorm to be installed. However, it is still possible to separate the two programs into different containers. Hence, the Project IceStorm FPGA tool chain is split into three containers. Since the only change done in this approach is the splitting of the container containing both Arachne-pnr and IceStorm, the Yosys image used in this approach is the same as in the previous microservice approach. Both the single-stage and multi-stage solution will be shown.

```
1   FROM ubuntu:14.04
2
3   RUN sudo apt-get update
4   RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
5                        gawk tcl-dev libffi-dev git mercurial graphviz   \
6                        xdot pkg-config python python3 libftdi-dev
7   RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
8   WORKDIR icestorm
9   RUN make -j$(nproc)
10  RUN sudo make install
11  WORKDIR /.
12  RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
13  WORKDIR arachne-pnr
14  RUN make -j$(nproc)
15  RUN sudo make install
16  WORKDIR /icestorm
17  RUN make -j$(nproc)
18  RUN make  uninstall
19  WORKDIR /.
20  RUN rm -f -r icestorm
```

FIGURE 4.6: Dockerfile for building the container with the Arachne-PNR tool

In Figure 4.6, it is shown how the Dockerfile is written in order to create a Docker image containing only Arachne-pnr. The method used is quite simple. Since the IceStorm files are only needed to install Arachne-pnr, one can simply install IceStorm first, then install Arachne-pnr and uninstall and remove IceStorm afterwards. This procedure can be seen in the Dockerfile, as the first 15 lines are identical to the first 15 lines in Figure 4.5. Then the next 4 lines uninstalls IceStorm, and then removes the files. One should notice that line 11 changes the working directory back to the root directory so that Arachne-pnr is not installed within the icestorm directory like in the monolithic approach. This would obviously not work, as the arachne-pnr directory would be removed along with the icestorm directory.



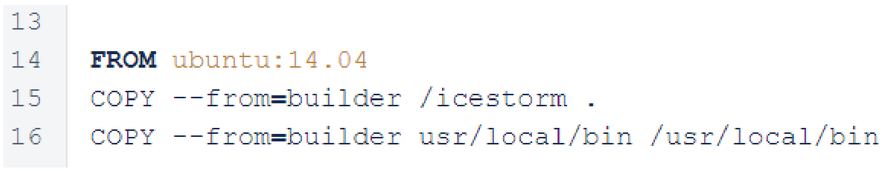
```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
```

FIGURE 4.7: Dockerfile for building the container with the IceStorm tools

Figure 4.7 shows the how the image containing only IceStorm was created. This was very simple as one only needed to install the IceStorm software in the exact same environment as in all of the previous images.

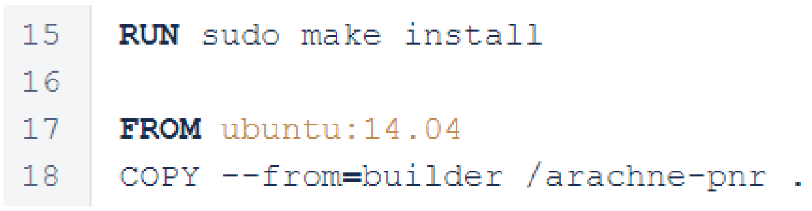
The multi-stage build technique was also applied to this these images. Also, due to how Docker images are built, as explained in Section 3.1, each new instruction executed in a Dockerfile adds another layer to the Docker image. Therefore, it can only increase in size, even if the files are removed from the image, as it is stored in the previous layer. The files that are removed will not be available in the container, but the information is stored in the image. Therefore, multi-stage builds will be extra important here, compared to the previous approach, to remove the unnecessary IceStorm files from the final image.



```

FROM ubuntu:14.04
COPY --from=builder /icestorm .
COPY --from=builder usr/local/bin /usr/local/bin
```

FIGURE 4.8: Dockerfile for building the container with the IceStorm tools, utilizing multi-stage build



```
RUN sudo make install

FROM ubuntu:14.04
COPY --from=builder /arachne-pnr .
```

FIGURE 4.9: Dockerfile for building the container with the Arachne-PNR tool, utilizing multi-stage build

Figure 4.8 and 4.9 shows how the multi-stage versions of the images were created. For simplicity, only the second build-stage in each Dockerfile is shown. In the IceStorm image, the icestorm directory, as well as the binary files created from the installation is created, while in the Arachne-pnr image, only the arachne-pnr directory is copied over.

4.4 Comparison of the Different Implementations

In this section, it will be examined how the three solutions compare to each other. It has already been verified through a simple test that the solutions work correctly, meaning that it has been verified that all the solutions can execute synthesis, place-and-route, and bitfile generation. Here, the solutions will be investigated further to discover if there are any potential differences in how the solutions scale. As this solution is intended to be deployed in the cloud and can potentially be used by a large number of people simultaneously, it is important that the solution has the ability to scale efficiently. This is the reason the tool chain was built in three different ways, so it could be evaluated whether a microservice solution was more effective in a cloud setting. A microservice architecture also provides more freedom when it comes to updating and maintaining the different components in the tool chain. However, when the tool chain is split up, it may introduce more overhead and complexity.

Section 2.4.7 introduces an objective function that aims to place a set of containers onto a set of VMs in the most efficient way possible. There are too many uncertainties regarding the underlying infrastructure to use this function directly in this thesis, and it is unlikely the cloud infrastructure provider, Digital Ocean, utilizes this specific function to handle VM and container placement. However, as explained in Section 2.4.7, the function consists of four terms, some of which can be analyzed qualitatively to see that there is a possibility that the microservice architecture will provide a more efficient solution.

In the second term of the function

$$\sum_{d \in D} \sum_{c_d \in C_d} \sum_{v \in V} \sum_{k_v \in K_v} \left((1 - z_{(d,k_v,t)}) \cdot (x_{(c_d,k_v,t)} \cdot \varDelta_d) \right).$$

$\varDelta_d$ refers to how long it takes to deploy a container. It is possible that smaller containers will deploy faster, hence saving costs. Furthermore, term three and four

$$\sum_{v \in V} \sum_{k_v \in K_v} \omega_f^R \cdot f_{(R,k_v,t)}$$

$$\sum_{d\in D}\sum_{c_d\in C_d}\sum_{v\in V}\sum_{k_v\in K_v}\left(\omega_s \cdot s_{(i,c_d,t)} \cdot x_{(c_d,k_v,t)}\right)$$

aims to minimize the amount of free resources on each VM, and allocate the minimal amount of resources to each container respectively. By splitting up the architecture into smaller images, it is possible have a higher utilization of each VM, and if the smaller containers can utilize less resources than the larger one, the benefit will be even greater.

To investigate the scaling capabilities of the solutions, a few parameters were analyzed. The first one is the size of the Docker images. As Docker images can be quite large, it is important to investigate if any of the solutions will take up less disk space than the others. Furthermore, it is interesting to see if the images which were created by utilizing multi-stage builds, yields any significant improvement to the image size. If the image size is reduced this will contribute to reduce the $f_{(R,k_v,t)}$ function, which sums up the free available resources. Hence the cost function will be decreased. Furthermore, if an image is smaller, it will take a shorter amount of time to fetch it onto a VM for the first time. Hence, $\mathit{\Delta}_d$ could also be reduced from smaller image sizes.

After investigating the image sizes, the performance of the containers was measured, with regards to execution time, CPU utilization and memory usage. Execution time means how long it takes to execute a given set of tasks. CPU utilization referrers to how much CPU time a given set of tasks need, or in other terms, how much of the CPU's capacity must be used to execute a given set of tasks. Memory usage means how much RAM that is used to execute a given set of tasks. It was first investigated how the execution time of the Yosys container compared to the monolithic container when performing synthesis, and the how Arachne-pnr container and the Arachne-pnr + IceStorm container compared to the monolithic when place-and-route was executed. This was done to see if there were any significant difference in the time it takes to create a container from a smaller image than from a larger image. If it turns out that, for example, the monolithic container utilizes more time than the Yosys container to start, execute synthesis, and then stop, it shows that $\mathit{\Delta}_d$ is smaller for the Yosys container, hence it is a more desirable choice. Furthermore, the execution time was measured for the containers when they were given restrictions on how much memory and CPU time they could use. This was done to further investigate if any of the containers, based on the smaller images, would require more or less memory and CPU time to get the same execution time as the monolithic container. If any of the containers that were executing the same task needed less memory and CPU time than the others, it would contribute to reduce the $f_{(R,k_v,t)}$ function.

Regarding the actual size of the containers, meaning the thin R/W-layer that is created for each container, which was explained in Section 3.1, the size for one such layer was in the range of a few hundred KB. Therefore, they are not included in any of these comparisons, as they are so small that they can be neglected.

4.4.1 Image Size

The first attribute of the architectures that was measured, was the image size. As explained in Section 3.1, a Docker image is only needed once to create multiple containers. However, to have enough computing power and memory for multiple users in a cloud environment, a solution like IDE8 will usually be run on multiple virtual machines. Each one of these machines would need a copy of the Docker image to create containers. Therefore, making images as small as possible will make the solution require less disk space, hence, scale better over multiple VMs. Furthermore, a single Docker image can be quite large. It is therefore, important to be aware of how much disk space a single Docker image require, so that IDE8 can choose the best underlying infrastructure. It is also, very interesting to see how the multi-stage builds compare to the single-stage builds and see how much overhead that was removed.

Also explained in Section 3.1, a Docker image consist of layers, and if multiple images have the same layers within them, the layer is only stored once. Therefore, if all containers are stored on the same machine, simply adding together the total size of each image would be an inaccurate way of measuring the total size of a solution. To measure the total image size correctly in this case, one has to identify what layers all containers in a solution have in common and be sure to only add them once to the overall sum.

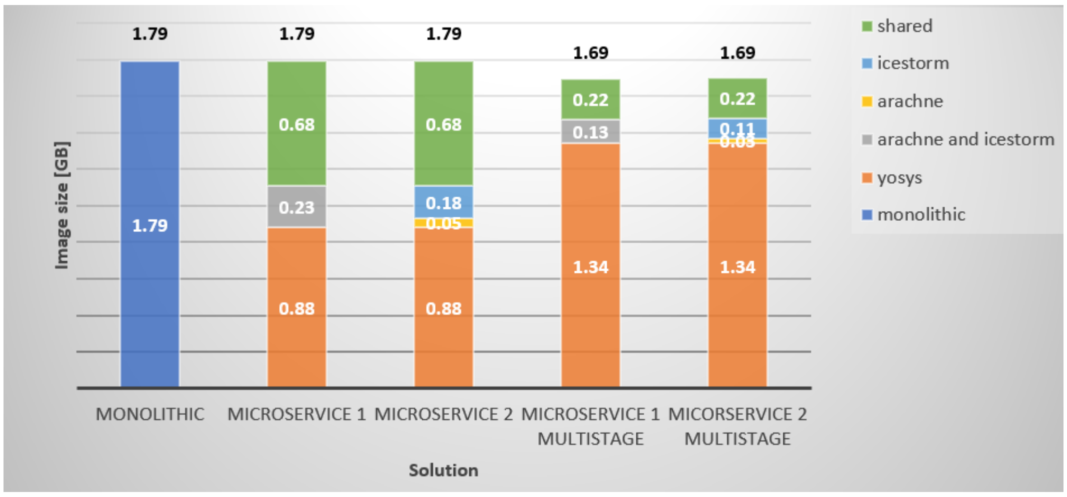


FIGURE 4.10: Overview of image sizes when stored on the same machine

Figure 4.10 shows how much disk space each of the solutions require when all the images of a solution is stored together. The figure shows that the monolithic, and the two single-stage microservice solution all use the same amount of disk space at 1.79 GB. This is not very surprising, as as the solutions consist of all the same components, and all have the same base-image. Both of the multi-stage builds have the same size, at 1.69 GB, about 100 MB less than the single-stage builds. Since the Yosys image did not utilize multi-stage, which was explained in Section 4.3.2, a lot of the advantages disappears, because all the installation tools that were needed, still exist in the Yosys image. If one were able to utilize multi-stage on the Yosys image, the overall disk space required, could probably be significantly reduced. This would also allow for the monolithic image to utilize multi-stage, which would mean that the total size of the monolithic image would be reduced to the same size as the other multi-stage solutions.

Furthermore, Figure 4.10 shows in detail, how many GB each individual image adds to the solution, and how many GB that is shared between all images. The single-stage solutions have a lot more shared layers than the multi-stage solutions. This is because the multi-stage Arachne-pnr and IceStorm images does not have all the packages that were installed so that the Project IceStorm software could be installed. In the multi-stage images these packages still exist in the Yosys image however. Therefore, the Yosys image appears to be much larger in the multi-stage solution than in the single-stage. Furthermore, it is clear that both the IceStorm tools and the Arachne-pnr tool does not require a lot of disk space compared to Yosys. In the multi-stage builds, IceStorm an Arachne-pnr only adds approximately 110 MB and 30 MB respectively.

For each layer in a Docker image to only be stored once, all the images of a solution have to be stored on the same machine. As stated previously, for a cloud solution with multiple users, there will probably be several machines in use at once. Therefore, it is meaningful to examine the total size of each Docker image. If, for example, a VM is only used for running Arachne-pnr, the machine would require all the layers of the Arachne-pnr image.

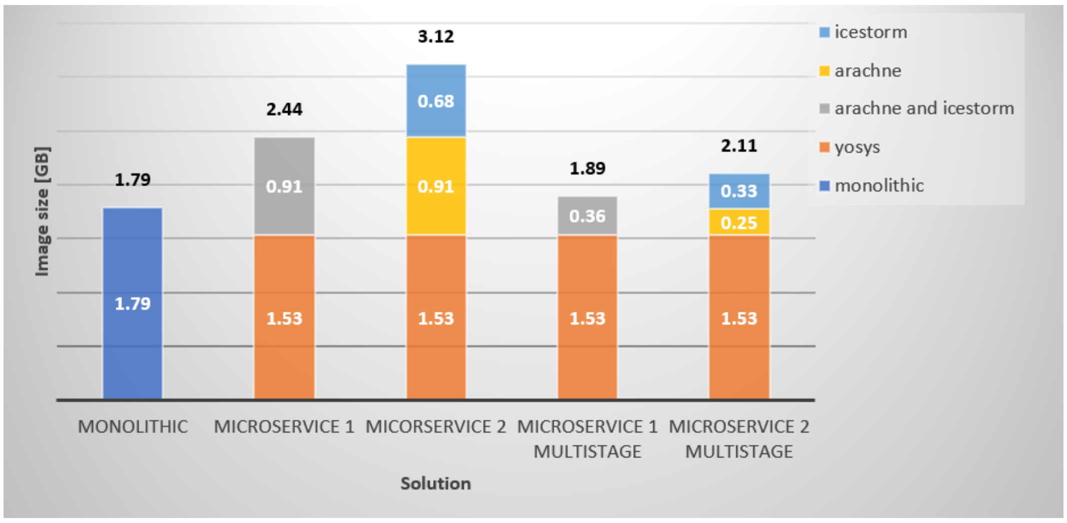


FIGURE 4.11: Total disk space required for each Docker image

Figure 4.11 shows the total image sizes for each solution. Each colored bar in this graph hence represent how much disk space is needed to store each one of the Docker images on separate machines.

To clarify how this actually works, a simple example will be used. If there, for example, are three VMs used to handle all the users of the tool chain, and it is assumed that one machine will handle all the syntheses, one will do the place-and-route, and on will do the bitfile generation. If the monolithic approach is used, each machine would need the monolithic image, hence requiring at total of $3 \times 1.79GB = 5.37GB$ of storage, while the "microservice 2" approach would only need $1.53GB + 0.91GB + 0.68GB = 3.12GB$ of storage. The multi-stage solution would require even less space as it would only require $1.53GB + 0.25GB + 0.33GB = 2.11GB$.

This example might not be the most realistic use-case, as stated in Section 2.5, synthesis is done a lot more, and as it will be shown later, require a lot more resources than, for example, bitfile generation. However, as shown in Figure 4.10, if each VM would require all images, it would still only require the same amount of disk space as the monolithic solution. Hence, the microservice architectures are always at least as good as the monolithic solution in terms of storage space.

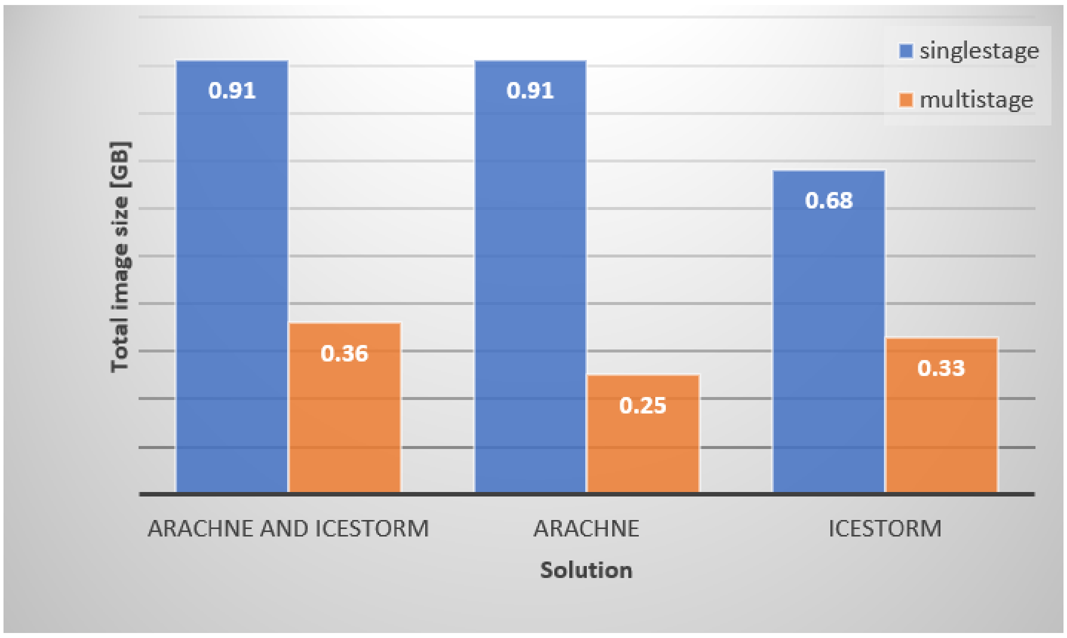


FIGURE 4.12: Plot of image sizes of the showing how single-stage builds compares to multi-stage builds

Figure 4.12 shows how the total size of the multi-stage builds compare to the single-stage builds, and Table 4.1 shows how much the image size was reduced. This shows that utilizing multi-stage builds can offer a significant improvement. Because of the way Docker images are built in layers, adding a new layer to the image cannot reduce the size of the image. This is best shown in the Arachne-pnr image, where both IceStorm and Arachne-pnr were installed, and then IceStorm was uninstalled. The image size however, remained the same as the container with both Arachne-pnr and IceStorm. Therefore, without multi-stage builds, this approach would be rather useless compared to the solution where IceStorm and Arachne-pnr were in the same image. However, with multi-stage, only the files needed can be copied over to the final image, allowing for even better memory utilization with a 3-container solution. If one looks at a similar example as the one earlier in this section, the 3-container solution would only need $0.25GB + 0.33GB = 0.58GB$ to utilize two VMs, where one is running Arachne-pnr and the other is running IceStorm. The solution with Arachne-pnr and IceStorm in the same container would need $2 \times 0.36GB = 0.72GB$ to do the same.

Table 4.1: Improvement for multi-stage build compared to single-stage

Image	Size reduction [%]
Arachne and IceStorm	60%
Arachne	73%
IceStorm	51%

There are two key results to take from these comparisons. The first one is that, utilizing multi-stage builds is a good approach for reducing image sizes, as shown here, the Arachne-pnr container benefited immensely from this, and got reduced by 73%. However, it is important to notice that how much multi-stage builds can help is extremely case specific. Therefore, one cannot make any claims as to how much multi-stage builds can help in general. It is only possible to say that, in this specific case, it was a very good approach.

The other result is that dividing the tool chain into several containers can result in more efficient use of disk space, which can reduce both the $f_{(R,k_v,t)}$ function and Δ_d. Because of how each layer in a Docker image is only stored once per machine, the solutions that were divided into two and three containers will never take up any more total disk space than the monolithic solution. Furthermore, as shown in the examples in this section, there are cases where the total disk size required would be reduced significantly by utilizing a microservice architecture when the solution is run on multiple VMs. This is consistent with Section 2.3, where it is stated that microservices scale more efficiently when deployed on multiple machines.

4.4.2 Performance

The image size is an important metric, as it is important to minimize the storage cost of a cloud solution. However, the image is only needed once to generate several containers, and disk space is not the most expensive resource. Therefore, measuring how much memory and CPU a container needs to function properly is arguably more important than the image size, since the goal of the cloud solution is to scale to several thousand users, and each user needs a separate container to run in his or her Workspace.

To get an overview of how the containers performed, first a simple test was performed. Each container used the rot example, which was mentioned in Section 4.3.1, and executed one or more of the instructions in Figure 4.2. For example, the monolithic container executed all the steps, the Yosys container executed the synthesis and so on. The "docker stats" command, explained in Section 3.1 was used to monitor the memory usage and CPU usage of the containers. The peak values of the memory and CPU usage was then noted. Furthermore, the memory usage when the containers were idle was also noted. The purpose of this test was only to get an understanding of what components in the tool chain that required the most computing resources, to evaluate if there were any components that could be ignored in future tests. As this test is quite inaccurate, and peak values can vary a lot, this test cannot be used to get any accurate difference between the different solutions.

Table 4.2: Results from running the rot example on each container

Container	CPU peak [%]	Memory Peak [MiB]	Default Memory
Monolithic	50	56.3	1 MiB
Yosys	40	7.5	1 MiB
Arachne + IceStorm	55	58.7	1 MiB
Arachne-pnr	50	55.5	1 MiB
IceStorm	6	1.0	1 MiB

Table 4.2 and 4.3 displays the results from running the rot example. From Table 4.2, it is clear that the Arachne-pnr tool is the most memory hungry, as all the containers that ran the place-and-route has a peak memory usage of over 55 MB, while the Yosys and IceStorm containers used very little memory.

When it comes to CPU utilization, it is clear that Arachne-pnr and Yosys both have relatively high CPU utilization. However, the CPU utilization is quite an unstable metric, which can vary quite a lot from each execution. Therefore, the main result to take away from this measurement, is that the IceStorm tool icepack utilizes much less CPU power than the other tools.

The last metric that is included here is the default memory usage, which refers to the memory a container utilizes when it is not executing any task. This could be an important metric, as a container would often be left running without doing anything for large amounts of time. So, if any of the containers had a much larger memory usage in idle mode than the others, this could possibly be an obstacle when the service is being used by multiple people. However, from the table it is clear that all the containers utilize about the same amount of memory when they are not being used. The memory usage in idle mode is also very low compared to the peak values.

These results could vary quite a bit as mentioned. Therefore, one should not put too much weight on the actual values. However, it is clear from these results that IceStorm's icepack tool is using an insignificant amount of CPU and memory compared to Arachne-pnr and Yosys. Therefore, when the next tests are conducted, the icepack tool will not be included.

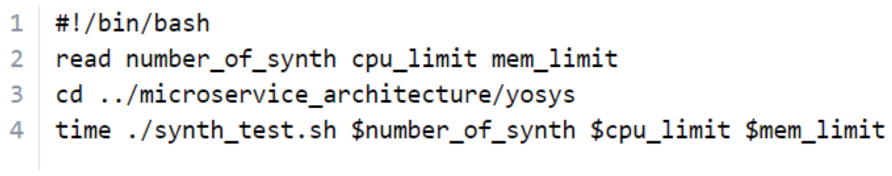
Table 4.3: Summary of results from running the rot example on each solution

Architecture	CPU peak [%]	Memory Peak [MiB]
Monolithic	50	56.3
Microservice 1	55	58.7
Microservice 2	50	55.5

Table 4.3 displays some of same data as in Table 4.2, except here the architectures are merged together, providing a more clear image of how each of the architectures performed on these metrics. There is little difference in memory usage and CPU usage between the solutions. Therefore, no conclusion can be made regarding which solution is better.

After conducting the initial tests of the containers, three new tests were conducted to further explore how the tools behaved under different constraints. This would provide more insight into whether there would be any difference between the solutions, with regards to performance, when they are scaled to be used by multiple users. Furthermore, in the previous test, it was discovered that the icepack tool utilized an insignificant amount of memory and CPU time, and will therefore, not be included in these tests.

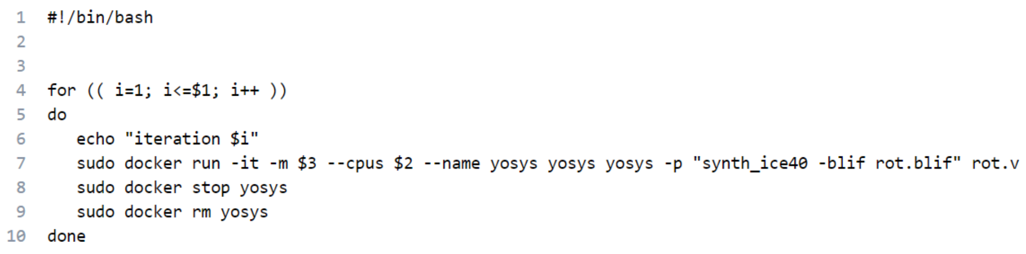
To achieve accurate time measurements, with as little possibility for interference and human error, the execution of the tests was automated with shell scripts.



```
1  #!/bin/bash
2  read number_of_synth cpu_limit mem_limit
3  cd ../microservice_architecture/yosys
4  time ./synth_test.sh $number_of_synth $cpu_limit $mem_limit
```

FIGURE 4.13: Top level test script for Yosys

Figure 4.13 shows the top level script used to execute the test for the Yosys container. The script for testing the monolithic container is identical to this one, except line 3 contains a different path to execute a different synth_test.sh script. These tests have three different parameters that are going to be examined, the number of syntheses or place-and-routes executed sequentially, and the amount of CPU time and memory the container is allowed to use. The script takes these values as input on line 2. Then, on line 4 the actual test-script is executed with the three parameters. The script is also executed with the "time" command, which is used to measure the execution time of the synth_test script.

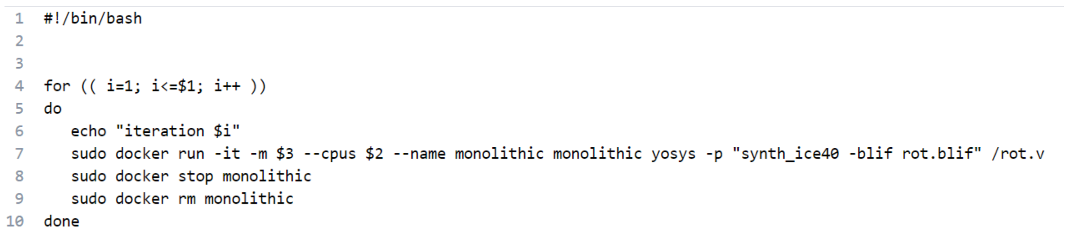


```
1   #!/bin/bash
2
3
4   for (( i=1; i<=$1; i++ ))
5   do
6       echo "iteration $i"
7       sudo docker run -it -m $3 --cpus $2 --name yosys yosys yosys -p "synth_ice40 -blif rot.blif" rot.v
8       sudo docker stop yosys
9       sudo docker rm yosys
10  done
```

FIGURE 4.14: synth_test.sh for yosys container

The test script for the Yosys container is shown in Figure 4.14, which referrers to the synt_test.sh script on line 4 in Figure 4.13. The script executes a routine, designed to examine the execution time of the Yosys container. The script starts up the Yosys container and executes synthesis a given number of times, which is specified in the script in Figure 4.13. The container is started with the "docker run" command on line 7. Several flags are used when starting the container. The -i flag is used to run the container in interactive mode, this connects the STDIN of the bash terminal window to the container, allowing one to view the output text from the container while it is running, making it easier to confirm that the container is executing its tasks correctly. It is also run with the –cpus and the -m flags, these are the flags used to specify the CPU- and memory limitations of the container. Also, the container is given the name yosys, making it easy to stop and remove the container after it has executed the synthesis.

The container is built from the Yosys image, described in Section 2.3. The container is also provided the command "yosys -p "synth_ice40 -blif rot.blif" rot.v", which is the command used to run the synthesis on the rot.v example. This command will execute immediately after the containers has become active. After the container has executed the synthesis, it is stopped and removed.



```
#!/bin/bash


for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name monolithic monolithic yosys -p "synth_ice40 -blif rot.blif" /rot.v
   sudo docker stop monolithic
   sudo docker rm monolithic
done
```

FIGURE 4.15: synth_test.sh for monolithic container

The test shown in Figure 4.15 is the equivalent test for the monolithic container. It performs the exact same routine as the previous script, but instead of running the Yosys container, it utilizes the monolithic container. These tests were made for all the containers, so that both synthesis and place-and-route could be run. Here, a full list of all test, is provided:

- **yosys_test**- executes synthesis with the Yosys container
- **monolithic_synth_test**- executes synthesis with the monolithic container
- **arachne_and_ice_test**- executes place-and-route with the Arachne-pnr + IceStorm container
- **arachne_pnr_test**- executes place-and-route with the Arachne-pnr container
- **monolithic_pnr_test**- executes place-and-route with the monolithic container

The first test aimed to investigate whether there was any overhead when starting a container from a larger image compared to a smaller image. However, containers are lightweight, and are supposed to have short startup and shutdown time. Therefore, to detect any overhead difference between the containers, they would have to be started and stopped multiple times. In this test, the execution time for the containers is measured when they are started, executes synthesis or place-and-route, then stopped, up to 250 times. This test is done for both synthesis and place-and-route. However, in the place-and-route test, the container containing both Arachne-pnr and IceStorm was not tested, as it would suffice to only compare the monolithic container and the Arachne-pnr container. This is because it is suspected that creating containers from larger images would have more overhead. Hence, it would be enough to only test the smallest and the largest image. As stated in Section 2.5, synthesis is typically performed a lot more than place-and-route. Therefore, the difference between the monolithic container and the Yosys container could be more important than the difference between the Arachne-pnr container and the monolithic container. However, the Yosys and monolithic container are quite similar in size. Therefore, the Arachne-pnr container is also tested, as it is much smaller than the others and could possibly provide clearer results. Table 4.5 shows a summary of which tests that were run and parameters that were used.

TABLE 4.4: Summary of what tests that were executed and the parameters used

Test	Number of iterations	Memory limit [MB]	CPU limit [%]
monolithic_synth_test	1-250	100	80
yosys_test	1-250	100	80
monolithic_pnr_test	1-250	100	80
arachne_pnr_test	1-250	100	80

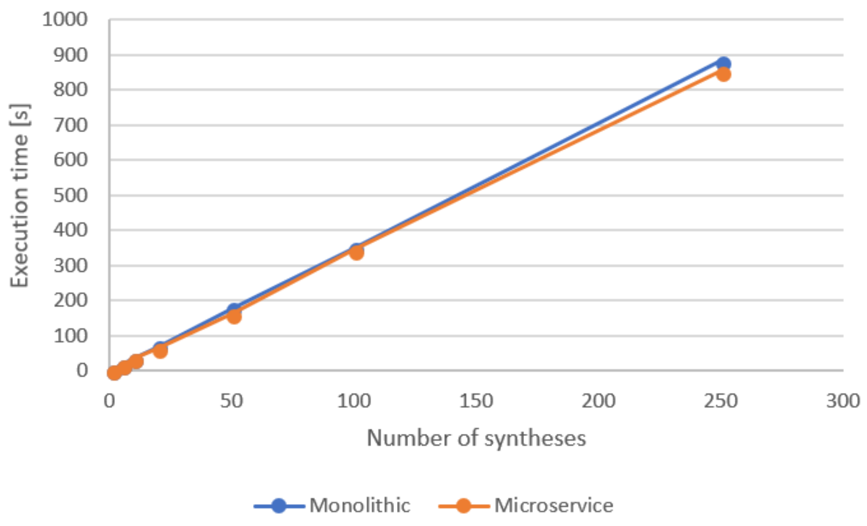


FIGURE 4.16: Results from running multiple syntheses

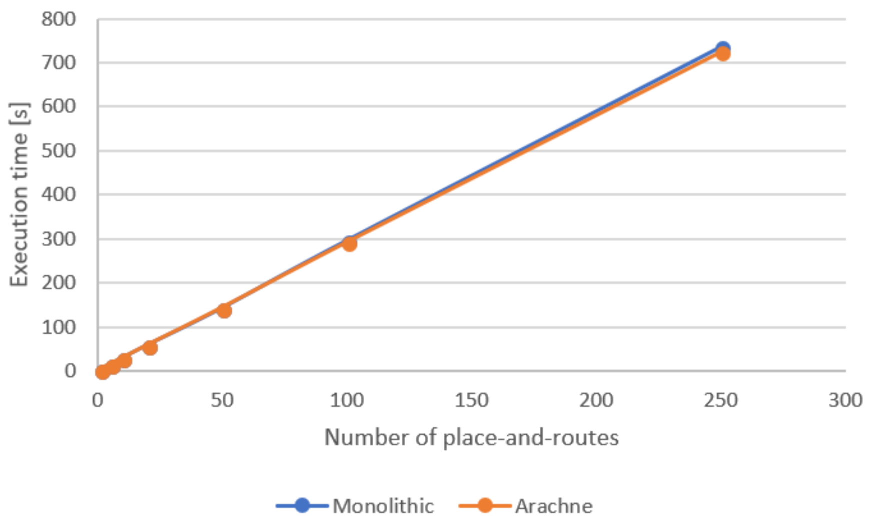


FIGURE 4.17: Results from running multiple place-and-routes

Figure 4.16 and 4.17 display the results from the multiple syntheses and place-and-route tests. As the graph shows, there are little difference between the execution time when utilizing the monolithic and the microservice containers. When 250 syntheses are executed, it does appear that the monolithic version is slightly slower than the microservice version, but this is probably due to small variations in the VM the tests were run in. So, it does not appear to be any difference in the time it takes to start and stop the Yosys and monolithic container or the monolithic compared to the Arachne-pnr container. Hence, if there is any overhead at all related to creating containers from larger images, it is much less than the execution time of the synthesis and place-and-rout of a very simple design and can therefore be ignored.

The two final tests measured and compared the memory- and CPU usage of each of the containers. These tests were conducted in order to examine whether the different solutions would perform differently given the same constraints. It could for example be the case that the monolithic container would require more memory and CPU time, as it is created from a larger Docker image than the container only containing Yosys. The previous test would not have detected this, as there were no strict constraints put on the container, which means that even though they were able to achieve the same execution time, they might have had different memory- and CPU usage. If this test is able to detect that any of the containers uses more time than the others, this could potentially mean that each instance of this container would require more resources when created in the cloud. Hence, the solution would require more resources, and therefore, be less scalable.

These tests will compare the execution time of the Monolithic container and Yosys container, when they perform synthesis on the rot example. Furthermore, the monolithic container, the Arachne-pnr container and the container containing both Arachne-pnr and IceStorm, will be compared when performing place-and-route on the rot example.

To execute these tests, a top-level test-script, like the one in Figure 4.13, was created for each container, which is shown in the list earlier. Furthermore, each container had its own version of the script in Figure 4.14 and 4.15, that were called in the top-level scripts.

First, the CPU utilization was varied, while the memory limit was kept constant. The memory limit was kept at 100 MB which is well above the peak values detected in Table 4.2. The CPU utilization was decreased in steps of 10 percentage points from 90% to 20% for the containers running place-and-route, and from 60% to 10% for the containers running synthesis. The reason these limits differ from each other, is that from Table 4.2, it appears that Yosys needs slightly less CPU time, and was therefore, not tested at any higher level than 60%, while Arachne-pnr needed more, and it was not deemed necessary to go all the way down to 10%. Table 4.5 shows a summary of which tests that were run and parameters that were used.

Table 4.5: Summary of what tests that were executed and the parameters used

Test	Number of iterations	Memory limit [MB]	CPU limit [%]
monolithic_synth_test	1	100	60-10
yosys_test	1	100	60-10
monolithic_pnr_test	1	100	90-20
arachne_and_ice_pnr_test	1	100	90-20
arachne_pnr_test	1	100	90-20

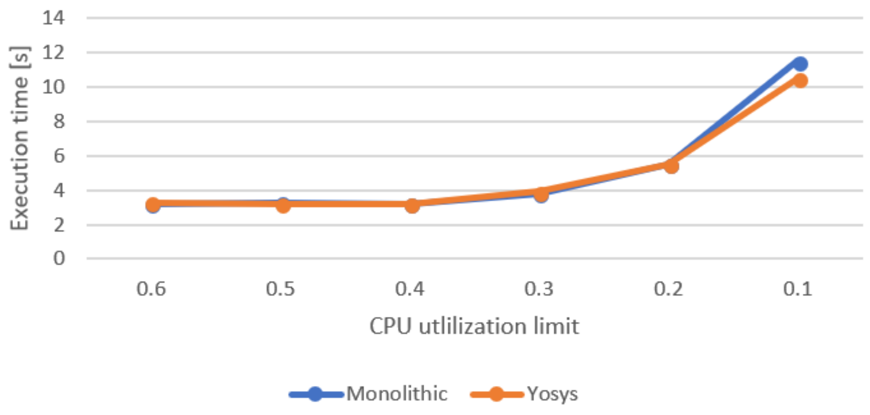


FIGURE 4.18: Results from changing the CPU utilization limit with synthesis

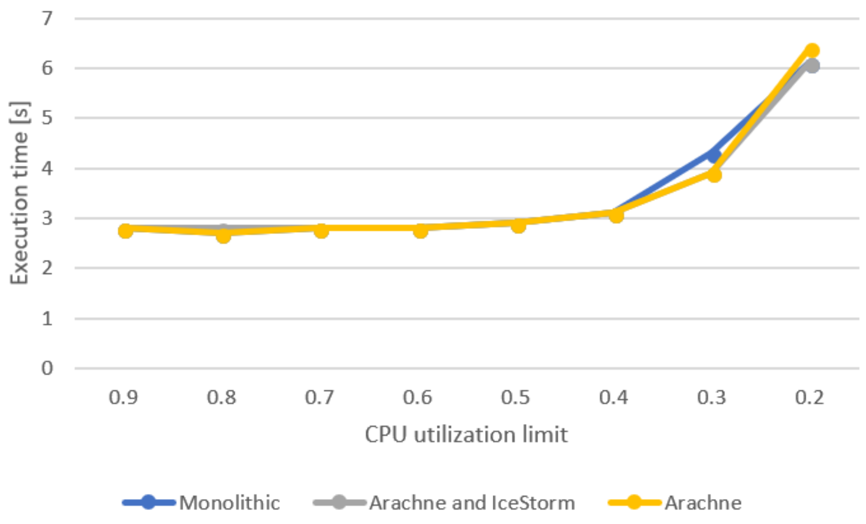


FIGURE 4.19: Results from changing the CPU utilization limit with place-and-route

Figure 4.18 and 4.19 shows the execution time of the synthesis and place-and-route as the CPU limit was changed. From both of the figures, it is clear that execution time for the different containers are almost identical. In Figure 4.18, the execution time of both the monolithic and the Yosys container start to increase when the CPU limit is at 30%, and they both increase quite similarly as the CPU limit goes down towards 10%. In Figure 4.19, all the containers start to increase in execution time when the CPU limit is at 40%. Also here, the execution time increases quite similarly. From these results, it appears to be no difference in how the containers behave, given the same CPU limitations.

In the final test, the CPU limit and number of iterations are kept constant. The CPU utilization limit is kept at 0.8 (80%), which one can see from the previous test, is high enough to not have any impact on the execution time. From the test results, shown in Table 4.2, one can see that the peak memory utilization of the containers is at approximately 55 MB for Arachne-pnr. Hence, it is expected that the execution time should start to increase around this value. The measurements for place-and-route were started with a limit of 70 MB and decreased in intervals of 10 MB down to 20 MB. The Yosys container had a very low memory peak, at under 10 MB. However, it is tested from 50 MB to 10 MB, as it is suspected that the memory usage could be higher. Table 4.5 shows a summary of which tests that were run and parameters that were used.

TABLE 4.6: Summary of what tests that were executed and the parameters used

Test	Number of iterations	Memory limit [MB]	CPU limit [%]
monolithic_synth_test	1	50-10	80
yosys_test	1	50-10	80
monolithic_pnr_test	1	70-20	80
arachne_and_ice_pnr_test	1	70-20	80
arachne_pnr_test	1	70-20	80

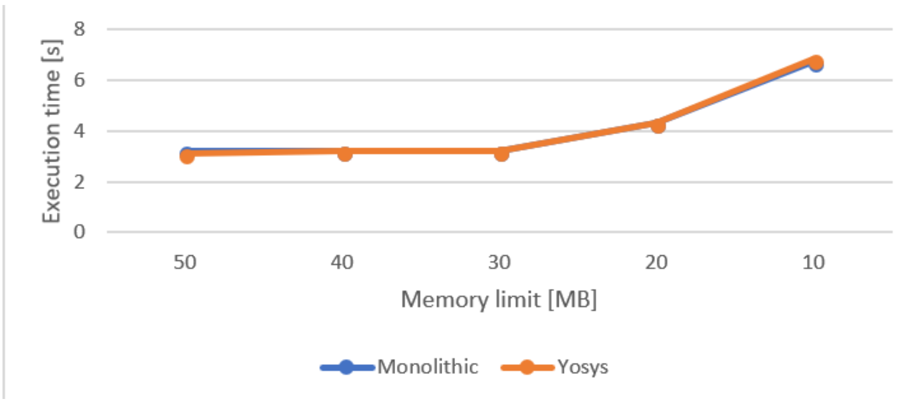


FIGURE 4.20: Results from changing the memory limit with synthesis

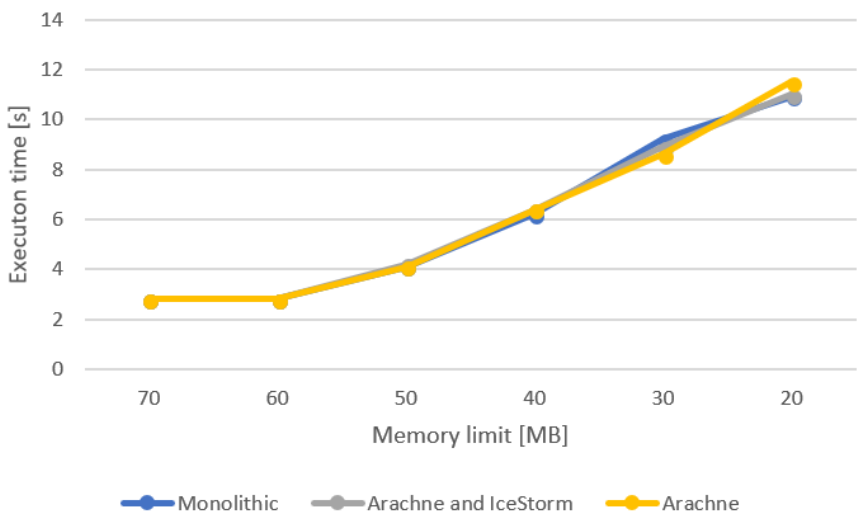


FIGURE 4.21: Results from changing the memory limit with place-and-route

The results of the final tests are plotted in Figure 4.20 and 4.21. As in the previous experiment, with CPU utilization, the execution time is almost identical for the containers as the memory limit is reduced. The execution time starts to change significantly when the limit is set at 50 MB for the place-and-route containers. These results are in line with the initial test conducted on the containers, where it was noted that the peak memory usage of all the different containers were a little over 50 MB. However, for the containers running synthesis, the execution time started to increase already at 20 MB, but also here, the containers are almost identical in terms of execution time. This experiment also indicates that there is no significant performance difference between the architectures. Therefore, based on these tests, no conclusion can be made regarding the different solutions ability to scale to multiple users.

To summarize the results from these test, it is quite clear that there is no reason to believe that taking the microservice approach will yield any performance and scaling improvements with regards to memory- and CPU usage. These experiments aimed to see if running containers from larger images would introduce any significant overhead with regards to memory- and CPU utilization. However, it is clear that this is not the case. If there is any overhead introduced by utilizing larger images, it is insignificant compared to the memory- and CPU usage of the FPGA tools within the containers.

4.4.3 Sources of Inaccuracy, Limitations and Challenges

The test and measurements presented in Section 4.4 provides a good understanding of how the different solutions performed, and if there would be any significant differences when scaling them to be used by multiple users simultaneously. However, there are still many untested scenarios regarding these architectures, some of which might provide valuable insight into the architectures, that has not been discovered in this thesis. This section will briefly cover some of the main sources of inaccuracy and limitations of the tests conducted here, as well as describing some possible additional tests that could be interesting to conduct. Furthermore, there were several challenges that arose while testing the solutions, it will also be explained what some of these were, as well as how they limited the examination of the different architectures done in this thesis.

There are a few sources of inaccuracy in these measurements. When it comes to image size, the different architectures were all implemented with the Ubuntu 14.04 base-image. However, when the image was created, the latest version of the Ubuntu image was always used. This image is constantly updated, and because the images were not all created at the same time, the base-images can vary slightly from each other. However, the base-image only varied with about 2 MB over the implementation period, so it is not a significant source of inaccuracy. Measuring the peak memory values and CPU utilization, can be quite inaccurate, and it may vary significantly. However, this measurement was only done to get an overview of how the containers performed, and both memory- and CPU utilization was investigated more in depth later. Therefore, no further action was taken to achieve more accurate results with this particular experiment. However, it is clear that the results displayed in Figure 4.16, 4.17, 4.18, 4.19, 4.20 and 4.21, are all susceptible to local variations, as there were small deviations in all measurements even in the flat areas of the graphs. These deviations are probably due to variations in background tasks being executed on the host desktop, but these deviations should be quite small and would not have been able to conceal any significant performance difference. Also, the flag -i that was used to attach STDIN to the container could also affect the run-time, as the container now has to print text. However, this operation should be insignificant to the actual synthesis or place-and-route, especially if the design that is used is bigger than the rot example.

Due to time limitations and some difficulties, there were some possible tests that could have been conducted in order to provide an even better understanding of how the containers functioned. One of the main limitations with all the container tests that were executed here was that only the small rot example was used. It would have been interesting to do the some of the same tests with a much larger FPGA design to see how this would have affected the execution time of the different architectures. However, as mentioned previously, it is unlikely that it would have yielded any different results with regards to how the containers compare to one another, as the other tests already showed that the monolithic container and the microservice containers had a similar startup and shutdown time. Therefore, testing a larger design would only have shown how the actual FPGA tools performed, and would not have provided any more insight into how the different containers behaved. However, it would have provided more insight into how much memory and CPU is needed to allocate to each container when run in the cloud. It would also have been valuable to test how the solutions actually scaled by creating multiple instances of the containers and having them running in parallel. This would have provided even more insight into how the different architectures scaled.

In Section 3.1, the scaling tool, Docker Swarm, was mentioned, which can create multiple Docker engines, so that multiple instances of the same images are available to create containers from. An experiment that should be executed in the future, to test further how the solutions actually scales, is to run multiple containers of the same type, for example the Yosys container, on the same VM. Then having the them all try to run synthesis in parallel and measure the execution time. This should be done with a varying number of containers until one begins to see significantly longer execution times. The same should be done for the monolithic container, and then, the execution times should be compared. This test would show if there is any difference in the performance reduction when multiple containers are run from the same image. This would show if there is any difference in how many container instances that can be run from the same image. If, for example, the monolithic image is able to have more containers running on top of it than the other containers, this could possibly negate the disk size benefits that the microservice architecture has, as one would need more copies of the images to achieve the same performance. However, there are not anything in the background material used in this thesis, that indicates that any of the containers would perform significantly better than the others on such a test, but nonetheless, it is still worth looking into.

As mentioned, there were a few challenges along the way, which limited the tests that could be conducted. When it came to a larger FPGA design, it proved to be difficult to find a design that was large, but at the same time simple enough in terms of file structure and modules, to actually use with the open source tools. One of the other major difficulties, was finding a good way of collecting and measuring the memory and CPU usage of the container. In the experiments that were conducted here, the data collection process was limited to "docker stats" and "time". However, it was attempted to utilize other tools to collect data in a more sophisticated manner. By utilizing a plugin from a service named SignalFx [71], it was possible to collect and plot real-time data from the containers. However, it was difficult to collect data with a high enough resolution for the results to be usable. Furthermore, the plugin used a lot of CPU and memory itself, which would have made it difficult to run multiple containers in parallel, and it would have introduced a great source of inaccuracy in the CPU measurements. Therefore, if one wished to test multiple containers at once, this would have to be run in, for example the cloud, but the time frame of the project did not allow for this to be done.

Chapter 5

IDE8 Integration and Evaluation of AWS FPGA Development

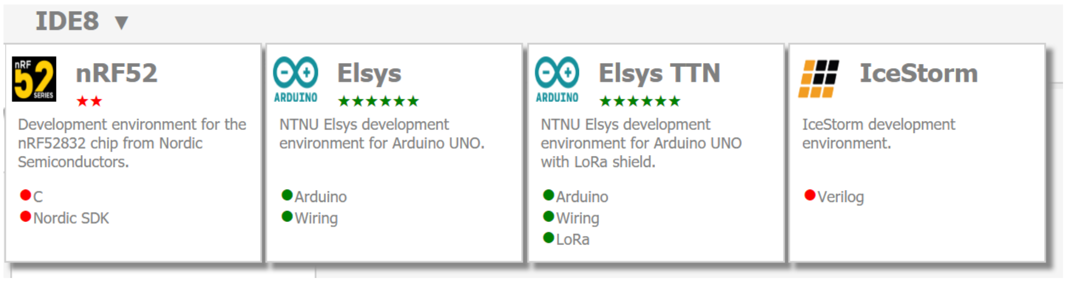
In this chapter, it will be looked into the IDE8 cloud solution. It will be shown how the solution from the previous section is integrated with IDE8, and how it can be integrated further with IDE8. Furthermore, the AWS FPGA developer AMI (Amazon Web Services FPGA developer Amazon Machine Image) is tested in this chapter.

5.1 Integrating Project IceStorm with IDE8

The solutions presented in the previous chapter was developed and tested locally on a desktop. However, once the initial design was finished, it was tested in the IDE8 environment. Because the stack had to be integrated with the backend system of IDE8, and this had to be done by an IDE8 developer, it was only attempted to integrate the monolithic solution with IDE8, and only basic functionality was added. How this was done is not included in this thesis, as it was done by others. In this thesis work, it was verified that the solution functioned properly, and that it was usable. Furthermore, some proposals were made regarding how to integrate the board agent with the tool chain.

5.1.1 Functionality

Here, it will be shown and explained the different features that are currently available with the IDE8 FPGA stack. All the Figures shown in this section are screen shots from the IDE8 website [72].



IDE8 ▼

nRF52
★★
Development environment for the nRF52832 chip from Nordic Semiconductors.
● C
● Nordic SDK

Elsys
★★★★★★
NTNU Elsys development environment for Arduino UNO.
● Arduino
● Wiring

Elsys TTN
★★★★★★
NTNU Elsys development environment for Arduino UNO with LoRa shield.
● Arduino
● Wiring
● LoRa

IceStorm
IceStorm development environment.
● Verilog

FIGURE 5.1: Available stacks in IDE8

Figure 5.1 displays the different development stacks that are currently available with IDE8. The IceStorm stack is the one developed in this thesis.

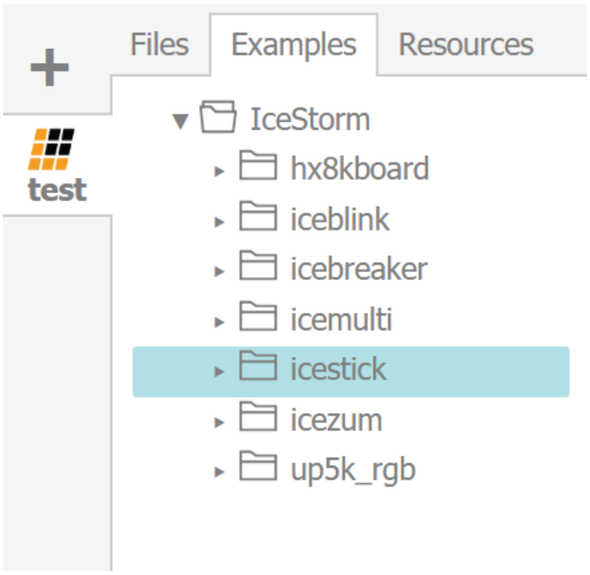


FIGURE 5.2: Available examples in the IceStorm stack

Figure 5.2 shows the different file directories available to the user. The “Files” directory and “Examples” directory were explained in Section 3.2. There is also a directory named “Resources”, but it is currently not used in the IceStorm stack. The figure also shows the examples that are available in the IceStorm stack. All the examples come from the IceStorm git repository and are made available for the user to experiment with.

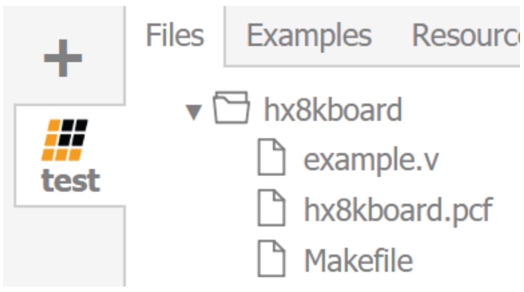


FIGURE 5.3: Files included in an example

In Figure 5.3, the hx8kboard example has been imported into the users file directory. This allows the user to use the files and perform operations on them. The example comes with a verilog file example.v, hx8kboard.pcf, which is the placement constraint file used to map outputs correctly on the HX8K development board, and a Makefile used to build the project.

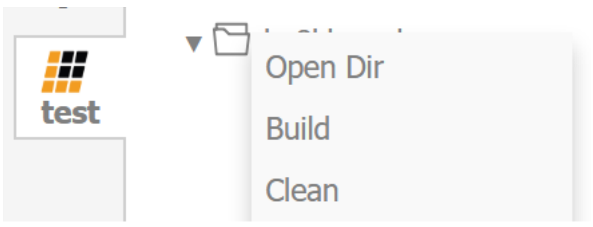
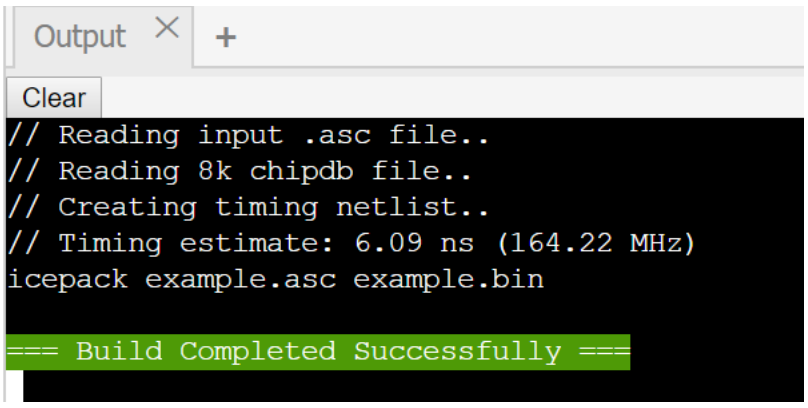


FIGURE 5.4: Example options

Figure 5.4 shows some of the options that are available when right clicking on the example. If one presses the "Build" option, the Makefile will be executed and the example will go through synthesis and place-and-route.

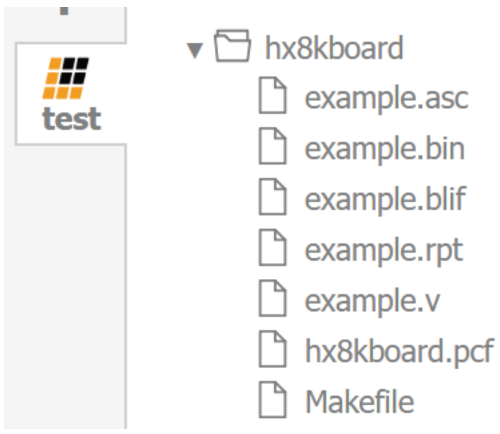


```
// Reading input .asc file..
// Reading 8k chipdb file..
// Creating timing netlist..
// Timing estimate: 6.09 ns (164.22 MHz)
icepack example.asc example.bin

=== Build Completed Successfully ===
```

FIGURE 5.5: View of the output terminal after running the build command

In Figure 5.5, it is shown the output window that appears once the build option is chosen. The output window shows all the output from all the tools used in the build process, such as the output information from Yosys and Arachne-pnr.



example.asc
example.bin
example.blif
example.rpt
example.v
hx8kboard.pcf
Makefile

FIGURE 5.6: All files generated from the build

After the build is finished, if one looks at the example folder again, one can see that all the files that were created during the build are available to the user. The files that are made available are, example.blif, which is the Yosys output, example.asc, which is the output from Arachne-pnr, example.bin, which is the binary file generated by icepack, and the example.rpt file which is a timing report generated by the icetime tool.

However, when a user is in the midst of a project, as stated in Section 2.5, one typically does not build the entire project, and one does a lot of quick testing that would be impractical to put in a Makefile. Hence, it is important to be able to interact with the individual tools. In IDE8, the user can open a bash terminal window, allowing the him or her to interact with the tools inside the container.

Some of the basic functionality however, is yet to be implemented into the IDE8 FPGA solution. The main part that is still missing is the integration with the Board agent, which allows one to connect and download the design onto an FPGA. This will however be explained in more detail in Section 5.1.2.

Even though the board agent does not support the IceStorm stack yet, there has already been set up a development board that is connected to IDE8.

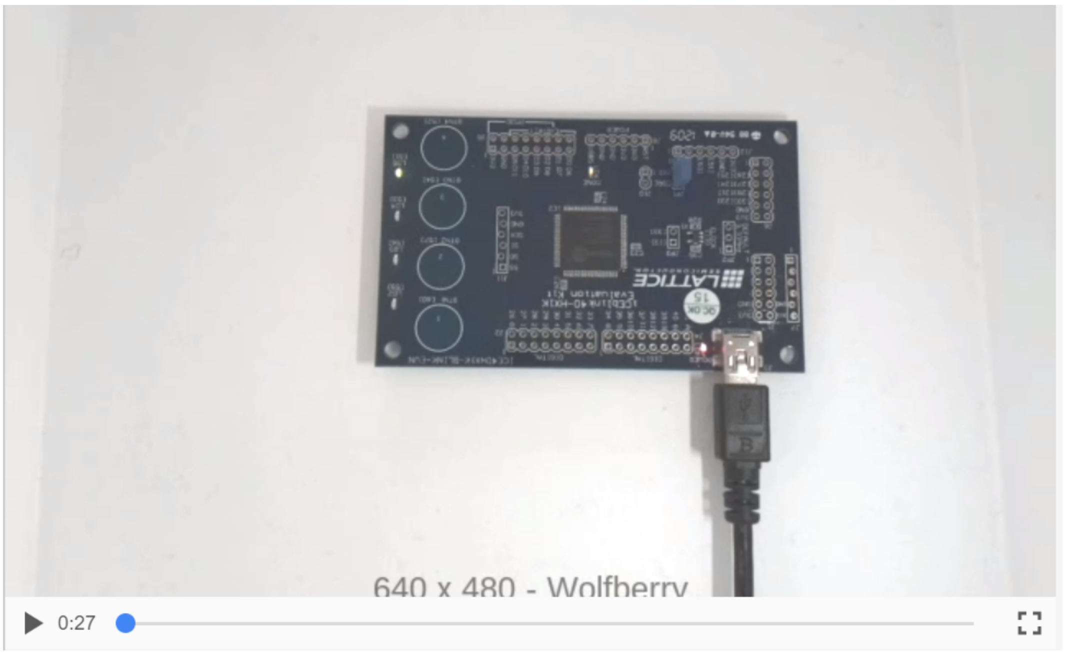


FIGURE 5.7: Live video feed of the hx8k board

Figure 5.7 shows the Lattice hx8k development board that is connected to a raspberry pi, at the IDE8 office. For all the stacks available, IDE8 has one or more devices connected to it, allowing users to test their design. Each device has a live video feed so that one can watch the development board, to, for example, verify that LEDs are blinking correctly.

5.1.2 Board Agent

An essential part of the FPGA development work flow is the possibility to test the design on an actual FPGA. This is one of the features IDE8 aims to offer. In the previous section, it was mainly focused on the Workspace Agent integration. In this section, integration with Board Agent will be discussed. As previously stated, this feature has not yet been implemented into IDE8.

There are three essential components involved in downloading code to an FPGA: the FPGA itself or development board, the binary design file and the iceprog program. Iceprog is the driver for a FTDI based programmer, mentioned in Section 3.5.

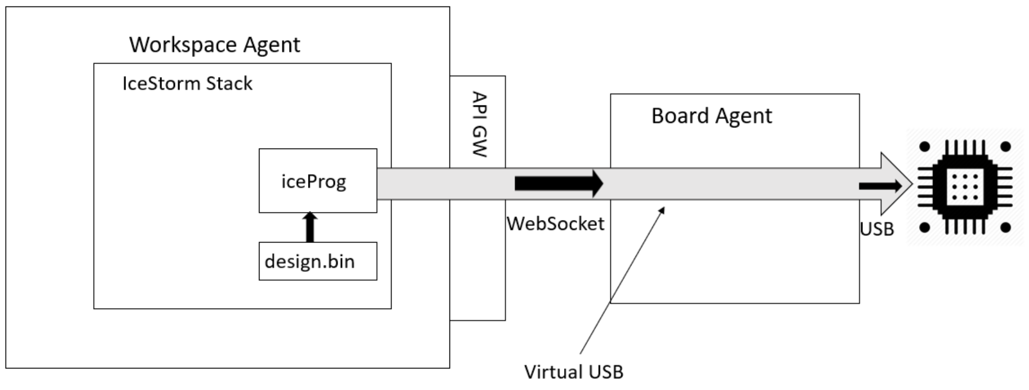


FIGURE 5.8: Overview of Board Agent connecting an FPGA to a Workspace

Figure 5.8 shows a diagram of how the programming of the FPGA could work. The iceprog driver will be installed in the Project IceStorm stack, regardless of which architecture from Chapter 4 that is chosen. The design.bin file is located in the users file directory, as shown in Figure 5.6. To connect to the FPGA development board, there will be a virtual USB connection from the FPGA to the Workspace agent. The virtual USB connection will go through the physical USB connection from the FPGA to the Board agent, which will expose the USB connection over the WebSocket to the Workspace agent, and hence, the iceprog driver. This method is quite similar to how other stacks connect to for example an Arduino, the main difference being that it is the iceprog driver that needs access to the development board, which is obviously not the case for an Arduino.

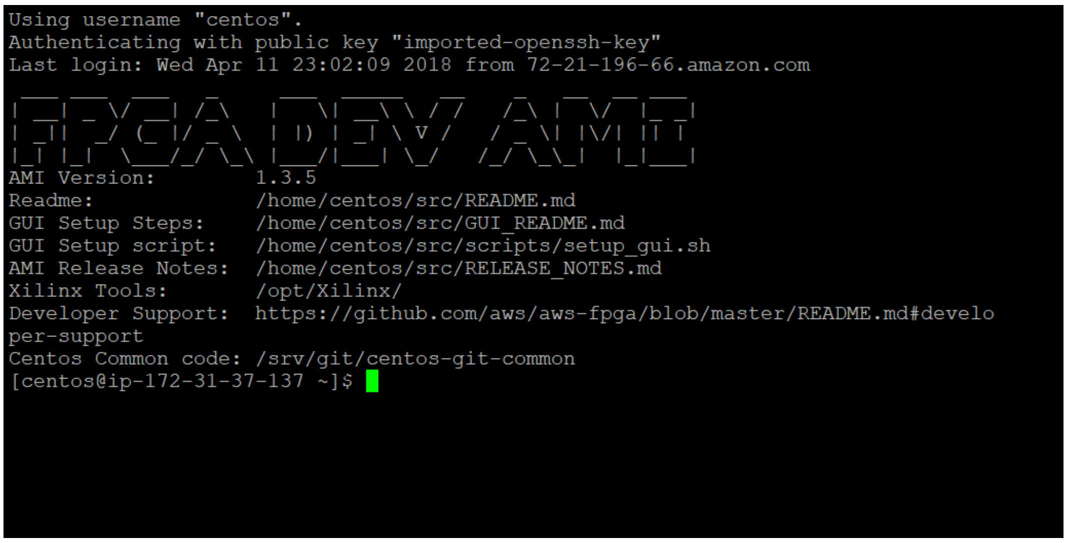
5.2 Testing The AWS FPGA Developer AMI

To get a better understanding of how the AWS (Amazon Web Services) FPGA developer AMI (Amazon Machine Image) compares to IDE8, getting hands-on experience was necessary. Hence, it was created an AWS account, where a VM with the FPGA developer AMI was set up. The AWS F1 instances, which were the virtual machine instances with FPGAs connected to them, were not tested, as it requires quite some work to create an FPGA design, and integrate it with the FPGA shell explained in Section 3.3. Furthermore, the main function of the F1 instances is FPGA utilization, meaning to use FPGAs in production to accelerate computations, and not development, meaning it is a feature that the IDE8 FPGA development tool chain does not aim to compete with, at least in the short term. How AWS and IDE8 compares will be discussed further in Chapter 6, this section will only explain the process of setting up the AWS FPGA developer AMI, and how the user experience was.

AWS mainly provides IaaS; hence the user has to manage his or her own infrastructure. When setting up the FPGA developer AMI, one has to go through the same steps to as when setting up any other VM. The first step is to choose the FPGA developer AMI, which is available on the AWS marketplace. Then one has to configure the desired VM. As explained in Section 3.3, the FPGA developer AMI is made by AWS, contains Xilinx Vivado, and there are no costs associated with utilizing the software. However, there may be costs associated with renting the underlying infrastructure. In this test, it was decided to try the only VM that could be used for free, the t2. micro, which has just 1 GiB of memory.

After one has chosen the AMI and what VM instance to run it on, there are a few steps that are needed to configure the VM. These steps are not exclusive to the FPGA developer AMI, but are general for all AWS VMs, hence they will not be explained in detail here. There are several settings for managing the security of the VM, as shown in Figure 2.3a in Section 2.4.6, when one is utilizing an IaaS, it is mainly the users responsibility to manage the security. However, AWS has several default security options, so that one can set up the VM without having to know the intricate details of the security settings. One also has to choose a storage medium, like an SSD disk, to store a snapshot of the AMI, so that the files created within the VM can be stored. A snapshot essentially stores the entire VM and its state, so that it can be easily accessed at all times. The disk size for the FPGA developer AMI has to be at least 70 GB. To have secure access to the VM one must also generate private keys, that are used to connect to it.

Once the VM is launched one can connect to it. To connect to the VM one has to use a tool named Putty [73].



```
Using username "centos".
Authenticating with public key "imported-openssh-key"
Last login: Wed Apr 11 23:02:09 2018 from 72-21-196-66.amazon.com
 ___ ___  ___   _     ___  _____   __    _   __  __ ___
| __| _ \/ __| /_\   |   \| __\ \ / /   /_\ |  \/  |_ _|
| _||  _/ (_ |/ _ \  | |) | _| \ V /   / _ \| |\/| || |
|_| |_|  \___/_/ \_\ |___/|___| \_/   /_/ \_\_|  |_|___|
AMI Version:        1.3.5
Readme:             /home/centos/src/README.md
GUI Setup Steps:    /home/centos/src/GUI_README.md
GUI Setup script:   /home/centos/src/scripts/setup_gui.sh
AMI Release Notes:  /home/centos/src/RELEASE_NOTES.md
Xilinx Tools:       /opt/Xilinx/
Developer Support:  https://github.com/aws/aws-fpga/blob/master/README.md#develo
per-support
Centos Common code: /srv/git/centos-git-common
[centos@ip-172-31-37-137 ~]$ 
```

FIGURE 5.9: Startup screen for the FPGA developer AMI

After one has connected to the VM using Putty and logged in, one is presented with the startup screen shown in Figure 5.9. Here, one is presented with the file locations of the installation scripts to install the Vivado GUI. The AMI already comes with the command line tools for Vivado pre-installed, but if one wish to utilize the GUI, one has to run setup_gui.sh

After the GUI was installed, it was needed to download the Xming tool [74] to be able to use the GUI on a Windows computer. The GUI is the same as if one were to utilize the already free Vivado software. However, to run the GUI on the t2. micro instance proved to be a slow and unpleasant experience. Therefore, if one wishes to use the Vivado GUI, one should use a more powerful VM instance. Other than that, to summarize, the user experience of the FPGA developer AMI was very similar to running Vivado locally on a desktop.

Chapter 6

Discussion

6.1 IDE8 vs Amazon and The FPGA Infrastructure Solution

There have been presented two other options other than IDE8 in this thesis, the AWS FPGA developer AMI and the FPGA infrastructure solution presented in Section 3.4. Here, it will be discussed what their differences are, and what advantages and disadvantages IDE8 has compared to them, and also if there are any features that could be taken from any of these solutions and be implemented into the IDE8 FPGA stack in the future.

Regarding the AWS solution, it is important to notice that it consist of two separate parts. The main part of the AWS solution is the AWS F1 instances, which provides a VM combined with high-performance FPGAs to be used to accelerate computational tasks. The other part is the FPGA developer AMI, which is an FPGA development environment that can be run on any AWS VM. As the IDE8 platform is mainly meant to be a development platform, the F1 instances are outside the scope of the IDE8 FPGA tool chain. Therefore, only the FPGA developer AMI will be discussed.

The FPGA infrastructure solution described in Section 3.4, is obviously IaaS (Infrastructure as a Service), and this goes for AWS as well. Both solutions will require the user to manage his or her own infrastructure. The IDE8 solution works more as a PaaS (Platform as a Service), implying that the underlying infrastructure is managed by the cloud provider. Managing the infrastructure offers a lot of flexibility, but also adds some complexity. For a hardware developer, configuring VMs may be somewhat unfamiliar territory, which may lead to inefficient use of resources. If one wishes to gain the full benefits of utilizing an IaaS, one should be able to utilize multiple types of VMs for different steps in the development process. One example could be to switch from a more general-purpose machine to a more compute optimized machine when one is ready to run synthesis and place-and-route on a design. Even if one is able to set up a simple VM and use it for FPGA development, it is a more difficult task to manage VMs in an effective manner, to optimize for cost and time for a project. For smaller projects, this extra layer of complexity and management may not be desired. The FPGA infrastructure solution aims to remove some of this complexity, by having the compilation and build of a project handled in the background. Hence, the user does not have to manage several VMs that are designed for different aspects of the design process, and only manage a single VM instance, which is the one used for writing code and doing behavioral verification. With the IDE8 solution, one does not have to manage any infrastructure. The development environment is available through a web browser, and the user's Workspace is instantly available once the user has logged in. This is one of the advantages of utilizing containers to create the Workspace, as they can be started much quicker than a VM. However, with this simplified process, compared to an IaaS solution, the user cannot manage how much computing power is allocated to perform task such as synthesis and place-and-route. Currently the limits of how much memory a container is allowed to use is set in the backend system. However, it would be possible to allow the user to set these limits, or at least present the user with some performance options, for example high, medium, or low speed synthesis. This would allow the user to get some of the same flexibility as with an IaaS solution without introducing any more complexity.

The AWS FPGA developer AMI provides Xilinx's Vivado tool for FPGA development, with several features. The FPGA infrastructure solution does not specify what tools or development environment that is going to be included in their solution. However, their general idea is that the developer should be able to choose from several tools from different FPGA providers. The IDE8 FPGA stack is based on the open source project, Project IceStorm. The FPGA development space is primarily dominated by closed source and proprietary technology. The two largest FPGA providers Intel and Xilinx, both have their own closed source tools, Quartus and Vivado. Creating open source tools for all FPGAs without the help of the FPGA providers is probably an impossible task. Project IceStorm was made possible by reverse engineering the bit stream of the Lattice iCE40 FPGA, which in and of itself is a complex task. When one also factors in that the Lattice FPGA was chosen specifically because it is one of the simpler FPGAs without a lot of custom functionality, it becomes clear that to do the same for e.g. the Xilinx UltraScale FPGA used in AWS's F1 instances is impossible to achieve, at least in a timely fashion. Hence, the tools available with AWS are a lot more to viable to be used, especially for the industry, compared to IDE8. For student use, and to gain development experience, the IceStorm tools provided with IDE8 may be sufficient. However, as stated earlier, the goal of this thesis was to get a working prototype, and it manly focused on the core elements of the FPGA development flow, namely synthesis, place-and-route and assembling. To get a more usable solution, more tools are needed, for example behavioral- and post-synthesis simulation tools. There exist open source tools for this, which are possible to integrate into IDE8.

Another type of tool that is included with Vivado, hence AWS, is Vivado HLS. It has become clear over the last few years that High-Level Synthesis is becoming more and more popular. Several large-scale projects utilize HLS and end up saving huge amounts of time and only losing a little bit of performance. So, it is necessary to ask whether traditional HDL development will cease to exist. For many commercial applications, it is probably true that the need for verilog and VHDL programmers will be significantly reduces as HLS tools keep improving. However, there is always going to be a need for someone to understand the underlying hardware, and especially for educational purposes traditional low-level programming will still be a vital part of the curriculum in order for students to gain an understanding of how digital circuits works. Hence, especially for student users, which is IDE8 main user group at the moment, the IDE8 FPGA stack does not need to include HLS tools to be relevant. However, there does exist several open source HLS tools that could be integrated with IDE8.

It is clear that it will be hard to compete in the FPGA development market with only open source tools. Therefore, another approach entirely for IDE8, is to have the FPGA tool providers build their own stacks in the IDE8 framework. This is a possibility that is already being looked into with other stacks within IDE8. This approach is somewhat similar to the one proposed with the FPGA infrastructure solution, where one can choose between different development environments. The approach would have to be modified, as IDE8 is web-based, while the FPGA infrastructure solution aims to provide the development environment through virtual desktops like AWS. The GUI components of the tools would have to be made compatible for web viewing, but several features of the GUI could be removed, as IDE8 already provides some of the features like an IDE and a file directory overview. How feasible a solution like this is, is unknown, but it could certainly provide a lot of benefits to the FPGA development process.

The final feature that separates the different solutions, is the ability to test the design on an actual FPGA. The closest one can get to this with AWS, are the F1 instances. However, they are not intended for testing, but for being used for computations. To utilize the F1 units one has to integrate the design with the AWS shell, found in the AWS FPGA HDK, to be able to communicate with the FPGA. For testing purposes, this is not a desired approach. One of the FPGA infrastructure solution s major proposals was that the cloud provider should invest into an FPGA farm that is available for the users to test their design. It is proposed that multiple types of FPGAs should be available, with different sizes and from different vendors. IDE8 utilizes the board agent, that will in the near future be able to connect a remote FPGA to the IDE8 platform. This feature allows the user to both access remote FPGAs managed by IDE8, or to download the agent to a desktop, and then connect a local device to the IDE8 platform. This method allows for a solution much like what the FPGA infrastructure solution aims to provide, IDE8 can connect and manage multiple devices and allow users to access them. As shown in Figure 5.7, IDE8 provides live video feed to the devices, so that visual tests, such as LED blinking, may be conducted. Each device that is connected to the IDE8 platform and is managed by IDE8 needs a raspberry Pi or something equivalent to run the board agent and the video feed. Since the board agent can run in such a lightweight environment, the costs of scaling to multiple devices is not very large. However, it remains untested, how well the entire solution would scale if one were to add hundreds, maybe thousands of devices, not just regarding FPGAs, but all types of devices. The current IDE8 architecture does probably not support the bandwidth required to have that many devices connected to it.

As stated in section 2.2, containers are a much more lightweight way of providing virtual computer environments than virtual machines. Because the stacks in IDE8 are container-based, it is possible to scale IDE8 to more users, and use a significantly smaller amount of storage space than with AWS. In Section 5.2, it was explained that one needs at least

70 GB of storage to store a snapshot of the users development environment (VM) in AWS. This means that for each user of the FPGA developer AMI, at least 70 GB of storage is required, even though AWS might have some backend solution to dynamically adjust how much storage a user actually needs. IDE8's web-based container solution will require much less than 70 GB. AWS's VM solution provides a separate copy of the Vivado tools for each user, while IDE8's container solution allows several users to share the same copy of the tools (Docker image). Even if multiple copies of the tools have to be made, the total size of the tools would still be a lot less than what is required with AWS. To estimate the storage space an additional user in IDE8 would require is difficult, as it has not been entirely decided which of the solutions from Chapter 4 that is going to be used. Furthermore, it is not clear what tools that are going to be added to the solution in the future.

To summarize, it is clear that the FPGA tools provided with the AWS FPGA developer AMI (Vivado), are a lot more powerful than the tool chain currently available in IDE8. However, in the future many of the features, such as HLS, can be offered in IDE8 with open source tools. For IDE8 to be able to keep up with the FPGA industry, proprietary closed source tools are probably needed. To achieve this, a collaboration with the FPGA vendors could be a possibility. It is clear that providing a web-based development environment for FPGAs has a lot of merit to it. It is significantly simpler for a user to get started developing than with a VM-based solution like AWS. Furthermore, IDE8's container-based solution will probably require less resources to facilitate a user base of a given size.

6.2 Which Architecture is Best Suited for IDE8

In Chapter 4, the Project IceStorm components were installed in Docker containers in three different ways.

The results obtained from measuring and analyzing the image sizes, clearly shows that splitting the tool chain into multiple containers is beneficial. Utilizing the 3-container solution will never cost more disk space than the 1-container solution, and in most cases, the 3-container solution would take up significantly less space. Furthermore, it was clear that utilizing multi-stage builds provided even more efficient disk utilization.

Hence, in terms of overall image size, it is clear that the 3-container solution, which utilizes multi-stage, is the better choice.

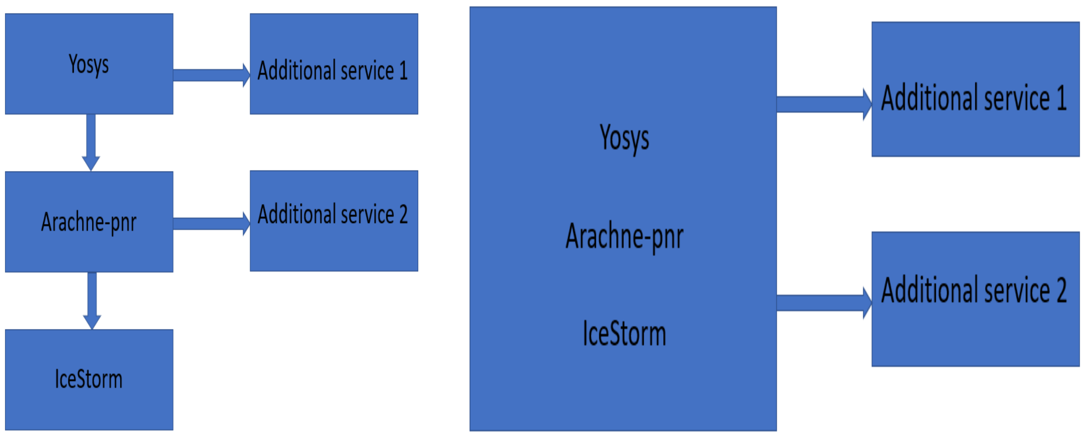
One test that was executed in this thesis in order to detect any performance difference between the solutions, was the test comparing the execution time of solutions when the number of syntheses and place-and-routes was increased. This test executed an increasing amount of syntheses and place-and-routes, from 1 to 250. What this test was designed to figure out was to see if there were any difference in execution time when a container was deployed from a smaller image, compared to the monolithic image. There were however, not discovered any significant differences in the results, hence showing that the 1-container solution did not perform any worse than the Yosys container or the Arachne-pnr container. This test shows that the startup and shutdown time of the containers are very similar. Hence, this test did not provide any clear arguments for or against any of the solutions. This was not too unexpected, as containers are supposed to be lightweight and should be able to start rapidly.

To further compare the different architectures, their CPU and memory utilization was measured, or rather how fast the containers were able to perform synthesis or place-and-route, given certain memory and CPU constraints. The goal of this test was to examine if there was any significant difference between the monolithic architecture and the microservice architecture. For example, if the monolithic container, made from a larger image, would require more memory and CPU than the microservice containers. From the tests conducted here, it was concluded that this is not the case, there were no clear trends that indicated that one of the solutions were more efficient than the other.

One potential drawback of a solution consisting of multiple containers is the additional complexity to the overall FPGA development solution. With a monolithic approach, all files generated inside a container, such as a blif file from Yosys, can be instantly used by the other tools inside the container. In a microservice solution, the file must be sent from the container to the users file directory, and then sent into the new container for further processing. With larger design files this could introduce some latency in the system. One example of how the microservice architecture can be more difficult to implement is the build functionality, shown in Figure 5.4. The build option finds the Makefile of a project and uses it to build the project. When all the tools used are in different containers, the build process becomes a lot more complex since IDE8 has to find the correct container for each instruction in the Makefile. However, during a FPGA design process, there are not done a lot of synthesis and place-and-route operations consecutively. Therefore, the overall impact this will have on a project is small. However, when more features are introduced, there might be more file transfers between containers, and if there is any noticeable latency in the file transfers, this may provide a worse user experience.

Even though there were no significant difference in the resource usage of the monolithic container compared to the other containers, there are several other benefits of utilizing a microservice approach compared to a monolithic approach. One of the main advantages is the flexibility one gets, as services can be added and removed without affecting the other services in the solution. However, it is very important to be aware of that even though the 1-container solution has been referred to as a monolithic solution, it could still be considered a microservice, as there is no strict definition of how small a microservice has to be. The 1-container solution can still have other microservices attached to it. In a more holistic view of IDE8, each stack within IDE8 can be seen as a microservice, regardless of how it is built up internally.

It is difficult to conclude which of the architectures that would be the optimal one to use in IDE8, as there were no scaling differences detected with regards to resource usage. With regards to image storage space, there is a clear benefit of utilizing a solution with multiple containers. It is furthermore, clear to see that in order for the FPGA tool chain to be usable, there has to be more features added. Hence, a microservice approach would be the best way to go, as it allows for simpler integration with other services later on. However, as mentioned previously, in a larger sense, there is no problem with defining the 1-container solution as a microservice, the service would just be a bit larger than in the case with 2 or 3 containers. This means that there are no problems with adding other microservices to the 1-container solution.



(A) (B)

Figure 6.1: Example of integrating both the 1- and 3-container solution with other microservices

Figure 6.1 shows a simple illustration of how additional microservices can be connected to both the 1- and 3-container solution. Therefore, the question becomes whether the tool chain needs to be split up into more than 1 container. As stated previously, one benefit of utilizing microservices, is to ability to add, remove and exchange services efficiently. Splitting the tool chain into three parts would allow for IDE8 to experiment further with for example other synthesis tools like Icarus Verilog. Furthermore, if a new place-and-route tool for other FPGAs is created, it would be much easier to include it in the tool chain. Also, considering the typical use patterns of FPGA tools, referring to that synthesis, place-and-route, and bitstream generation is rarely executed in quick succession, it is unlikely that any performance improvement gained from not having to transfer files between containers would come into play very often. The advantages regarding flexibility with further development of the FPGA tool chain, combined with the improved utilization of disk space that the smaller images provide, makes it clear that the 3-container solution is the best choice going forward.

Chapter 7

Conclusion

In this thesis, it has been developed an FPGA tool chain by using Docker containers, allowing it to be integrated with the IDE8 developer framework. Three different configurations of the tool chain were built and compared with regards to image size, and execution time with different CPU and memory constraints. The results show that the solutions consisting of more containers can utilize less disk space due to how layers in Docker images are only stored once. Furthermore, by utilizing multi-stage builds, the image size of the Arachne-pnr and IceStorm containers was reduced by 73% and 51% respectively. Furthermore, the container containing both Arachne-pnr and IceStorm was reduced in size by 60%. The results obtained from testing the CPU and memory utilization of the containers however, shows that there is no significant difference in how much resources the different solutions require.

There are several other advantages of a microservice approach compared to a monolithic approach. By utilizing a microservice approach, it becomes a lot simpler to add new services. The FPGA tool chain built in this thesis is far from complete, several features needs to be added. Therefore, a microservice solution will be the preferred approach. However, as stated earlier, the 1-container solution can still be considered a microservice, so it is still not clear that dividing the tool chain built in this thesis is favorable. However, running the synthesis, place-and-route and assembling in different containers will provide increase freedom in interchanging the components. There has not been done any comparison of different open source tools in this thesis. It is therefore possible that one wishes to for example switch Yosys with Icarus Verilog at some point. This would be a lot easier and can be done quite seamlessly if the components are run in separate containers. Therefore, it is concluded with that splitting up the tool chain into 3 containers would be the best solution going forward, as the flexibility of running the components in separate containers, combined with the improved disk space utilization, is a significant advantage.

In this thesis, it was also looked into other cloud-based FPGA tools, primarily the AWS FPGA developer AMI and F1 instances. Also, another paper [12] was discussed, as it proposed some alternative ways of offering FPGA development in the cloud. However, it is clear that IDE8 can offer a very different way of developing FPGAs in the cloud, as it is a web-based PaaS solution compared to the VM-based IaaS solution that AWS offers. The IDE8 board agent is also a good way of allowing users to utilize both their own FPGAs, as well as IDE8's FPGAs. However, it remains to be tested how this board agent scales, if IDE8 wishes to offer a larger number of devices. It is also concluded that if IDE8 wishes to compete with the industry, open source tools are rather limited, and a cooperation with FPGA tool providers could be one way of offering better FPGA tools.

7.1 Future Work

There have been mentioned a few components in this thesis that can be used in the tool chain, like debugging and verification tools. These are essential tools for FPGA development. However, here we have only focused on implementing a solution with the main components: synthesis, place-and-route and assembling. To further develop this solution, and make it more of a viable product, it should be investigated further if these tools, or if there exist other tools, that could be integrated with this solution.

In this thesis, only the Ubuntu 14.04 base image was used for the different Docker configurations. With its size of approximately 220 MB it generates introduces quite a bit of overhead. On docker hub, there exist an abundance of various Linux images, some which are considerably more lightweight than the image used here. If one were to find a smaller image that still contains all the necessary components for the IceStorm tools to function, this could significantly reduce the overall image sizes.

As mentioned in Section 4.4, the tests done in this thesis has some limitations with regards to measuring the actual scalability of the architectures. Hence, conducting tests where multiple containers are run in parallel and the memory usage and CPU usage is along with execution time is measured, would provide a much clearer view of how the different solutions scales and could possibly discover bottlenecks that has not been thought of or detected in this thesis.

Furthermore, to build a complete solution, the Board Agent should be integrated with the IceStorm Stack. This would be a very important step in providing a complete FPGA tool chain solution. However, as with most of the tool chain integration with IDE8, this is a job that probably has to be done by the IDE8 developers. Furthermore, the 1-container solution that is currently used in IDE8, should be swapped out with the 3-container solution.

Appendix A

Dockerfiles and Shell Scripts

This appendix includes all the Dockerfiles and shell scripts created and used in this thesis. The repository containing all the code is available at [2]. However, the repository is a little difficult to navigate and contains several branches, and several unused files. Therefore, the code that was actually used is included here.

Note that the variable "number_of_synth" in the top-level tests is also used in the tests that execute place-and-route. The variable just sets the amount of iterations, regardless of what operation is executed within the for-loop.

Monolithic Dockerfile:

```
FROM ubuntu:14.04


RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR arachne-pnr
RUN make -j$(nproc)
RUN sudo make install
RUN git clone https://github.com/cliffordwolf/yosys.git yosys
```

```
WORKDIR yosys
RUN make -j$(nproc)
RUN sudo make install
```

Yosys Dockerfile:

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/yosys.git yosys
WORKDIR yosys
RUN make -j$(nproc)
RUN sudo make install
```

Arachne + IceStorm single-stage Dockerfile:

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR arachne-pnr
RUN make -j$(nproc)
RUN sudo make install
```

Arachne + IceStorm multi-stage Dockerfile:

```
FROM ubuntu:14.04 as builder

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
WORKDIR /
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR /arachne-pnr
RUN make -j$(nproc)
RUN sudo make install



FROM ubuntu:14.04
COPY --from=builder /icestorm /icestorm
COPY --from=builder /arachne-pnr .
COPY --from=builder /usr/local/bin/i* /usr/local/bin/
```

Arachne single-stage Dockerfile:

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
WORKDIR /.
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR arachne-pnr
RUN make -j$(nproc)
RUN sudo make install
WORKDIR /icestorm
RUN make -j$(nproc)
RUN make  uninstall
WORKDIR /.
```

```
RUN rm -f -r icestorm
```

Arachne multi-stage Dockerfile:

```
FROM ubuntu:14.04 as builder

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
WORKDIR /.
RUN git clone https://github.com/cseed/arachne-pnr.git arachne-pnr
WORKDIR arachne-pnr
RUN make -j$(nproc)
RUN sudo make install

FROM ubuntu:14.04
COPY --from=builder /arachne-pnr .
```

IceStorm single-stage Dockerfile:

```
FROM ubuntu:14.04

RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install
```

IceStorm multi-stage Dockerfile:

```
FROM ubuntu:14.04 as builder
```

```
RUN sudo apt-get update
RUN sudo apt-get -y install build-essential clang bison flex libreadline-dev \
                     gawk tcl-dev libffi-dev git mercurial graphviz   \
                     xdot pkg-config python python3 libftdi-dev
RUN git clone https://github.com/cliffordwolf/icestorm.git icestorm
WORKDIR icestorm
RUN make -j$(nproc)
RUN sudo make install



FROM ubuntu:14.04
COPY --from=builder /icestorm .
COPY --from=builder usr/local/bin /usr/local/bin
```

Monolithic synthesis top-level:

```
#!/bin/bash
read number_of_synth cpu_limit mem_limit
cd ../monolithic_architecture/
time ./synth_test.sh $number_of_synth $cpu_limit $mem_limit
```

Monolithic synthesis test:

```
#!/bin/bash



for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name monolithic monolithic \
            yosys -p "synth_ice40 -blif rot.blif" /rot.v
   sudo docker stop monolithic
   sudo docker rm monolithic
done
```

Yosys top-level:

```
#!/bin/bash
read number_of_synth cpu_limit mem_limit
```

```
cd ../microservice_architecture/yosys/
time ./synth_test.sh $number_of_synth $cpu_limit $mem_limit
```

Yosys test:

```
#!/bin/bash



for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name yosys yosys \
            yosys -p "synth_ice40 -blif rot.blif" rot.v
   sudo docker stop yosys
   sudo docker rm yosys
done
```

Monolithic pnr top-level:

```
#!/bin/bash
read number_of_synth cpu_limit mem_limit
time ./pnr_test_3.sh $number_of_synth $cpu_limit $mem_limit
```

Monolithic pnr test:

```
#!/bin/bash



for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name monolithic monolithic \
            arachne-pnr -d 1k -p /rot.pcf /rot.blif -o rot.asc
   sudo docker stop monolithic
   sudo docker rm monolithic
done
```

Arachne and IceStorm top-level:

```
#!/bin/bash
```

```
read number_of_synth cpu_limit mem_limit
time ./pnr_test_2.sh $number_of_synth $cpu_limit $mem_limit
```

Arachne and IceStorm test:

```
#!/bin/bash



for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name arachne_pnr arachne_and_ice_multi \
            arachne-pnr -d 1k -p /rot.pcf /rot.blif -o rot.asc
   sudo docker stop arachne_pnr
   sudo docker rm arachne_pnr
done
```

Arachne top-level

```
#!/bin/bash
read number_of_synth cpu_limit mem_limit
time ./pnr_test_1.sh $number_of_synth $cpu_limit $mem_limit
```

Arachne test:

```
#!/bin/bash



for (( i=1; i<=$1; i++ ))
do
   echo "iteration $i"
   sudo docker run -it -m $3 --cpus $2 --name arachne_pnr arachne_pnr_multi \
            arachne-pnr -d 1k -p /rot.pcf /rot.blif -o rot.asc
   sudo docker stop arachne_pnr
   sudo docker rm arachne_pnr
don
```

Bibliography

[1] Kristian Aalde. Cloud based toolchain for hardware development. 2017. December. Project report. NTNU.

[2] Kristian Aalde. Ide8 fpga stack. URL https://bitbucket.org/KristianAalde/ide8-fpga_stack. bitbucket repository containing the code of this project.

[3] Jose De la Rosa and Kent Baxley. Lxc containers in ubuntu server 14.04 lts. URL http://en.community.dell.com/techcenter/os-applications/w/wiki/6950.lxc-containers-in-ubuntu-server-14-04-lts. Last visited: 06.02.18.

[4] Microsoft. Why a microservices approach to building applications?, 2018. URL https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices. Last visited: 20.02.18.

[5] Xilinx. Fpga vs. asic design flow, December 2017. URL https://www.xilinx.com/video/fpga/fpga-vs-asic-design-flow.html.

[6] Rick Mak. Dockerizing our python stack. URL https://code.oursky.com/dockerizing-our-python-stack/.

[7] Docker. About storage drivers, . URL https://docs.docker.com/storage/storagedriver/#container-and-layers. Last visited: 31.05.18.

[8] Docker. Use multi-stage builds, . URL https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds. Last visited: 03.05.18.

[9] Jon Anders Haugum. Ide8 architecture, 2018. Personal communication.

[10] Amazon Web Services. Aws shell interface specification, . URL https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md. Last visited: 02.05.18.

[11] Amazon Web Services. Aws fpga: Programmer's view of the custom logic, . URL https://github.com/aws/aws-fpga/blob/master/hdk/docs/Programmer_View.md. Last visited: 02.05.18.

[12] Laurentiu A Dumitru, Sergiu Eftimie, and Ciprian Racuciu. Using clouds for fpga development-a commercial perspective. *Journal of Information Systems & Operations Management*, page 42, 2017.

[13] Clifford Wolf and Mathias Lasser. Project icestorm. URL http://www.clifford.at/icestorm/.

[14] David Pellerin. Fpga developer ami. URL https://www.slideshare.net/AmazonWebServices/deep-dive-on-amazon-ec2-f1-instance-may-2017-aws-online-tech-talks. Last visited: 02.05.18.

[15] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[16] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 257–261. IEEE, 2008.

[17] EN Preeth, Fr Jaison Paul Mulerickal, Biju Paul, and Yedhu Sastri. Evaluation of docker containers based on hardware utilization. In *Control Communication & Computing India (ICCC), 2015 International Conference on*, pages 697–700. IEEE, 2015.

[18] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. URL https://martinfowler.com/articles/microservices.html. Last visited: 11.04.18.

[19] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[20] Amazon Web Services. Cloud solutions, . URL https://aws.amazon.com/solutions/?nc2=h_ql_s&awsm=ql-2. Last visited: 04.06.18.

[21] VV Arutyunov. Cloud computing: Its history of development, modern state, and future considerations. *Scientific and Technical Information Processing*, 39(3):173–178, 2012.

[22] John L. Hennessy and David A. Patterson. *Computer architecture A quantitative approach*. Morgan Kaufmann, 5 edition, 2012.

[23] Amazon Web Services. Announcing amazon elastic compute cloud (amazon ec2) - beta, August 2006. URL https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[25] Microsoft. What is azure stack? URL https://azure.microsoft.com/nb-no/overview/azure-stack/. Last visited: 30.05.18.

[26] Apache cloudstack. URL https://cloudstack.apache.org/. Last visited: 30.05.18.

[27] Technopedia. Software stack. URL https://www.techopedia.com/definition/27268/software-stack. Last visited: 31.05.18.

[28] Sumant Ramgovind, Mariki M Eloff, and Elme Smith. The management of security in cloud computing. In *Information Security for South Africa (ISSA), 2010*, pages 1–7. IEEE, 2010.

[29] Rackspace. The difference between private and public cloud, December 2017. URL https://www.rackspace.com/cloud/cloud-computing/difference.

[30] Michael J Kavis. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014.

[31] Microsoft. What is a private cloud?, November 2017. URL https://azure.microsoft.com/en-us/overview/what-is-a-private-cloud/.

[32] Beth Pariseau. Code spaces goes dark after aws cloud security hack, November 2017. URL http://searchaws.techtarget.com/news/2240223024/Code-Spaces-goes-dark-after-AWS-cloud-security-hack.

[33] Jon Brodkin. Gartner: Seven cloud-computing security risks. *Infoworld*, 2008:1–3, 2008.

[34] Prasad Saripalli and Ben Walters. Quirc: A quantitative impact and risk assessment framework for cloud security. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 280–288. Ieee, 2010.

[35] Yvo Desmedt. *Man-in-the-Middle Attack*, pages 759–759. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_324. URL https://doi.org/10.1007/978-1-4419-5906-5_324.

[36] 9 'worst practices' to avoid with cloud computing, November 2017. URL https://www.forbes.com/sites/joemckendrick/2014/01/29/9-worst-practices-to-avoid-with-cloud-computing/#37356db3378c.

[37] 2nd watch cloud security survey: Shared responsibility model confuses cloud customers, Sep 28 2017. URL https://search.proquest.com/docview/1943524163?accountid=12870. Copyright - Copyright NASDAQ OMX Corporate Solutions, Inc. Sep 28, 2017; Last updated - 2017-09-28.

[38] Amazon Web Services. Shared responsibility model, November 2017. URL https://aws.amazon.com/compliance/shared-responsibility-model/.

[39] Microsoft. What does shared responsibility in the cloud mean?, November 2017. URL https://blogs.msdn.microsoft.com/azuresecurity/2016/04/18/what-does-shared-responsibility-in-the-cloud-mean/.

[40] Andreas Berl, Erol Gelenbe, Marco Di Girolamo, Giovanni Giuliani, Hermann De Meer, Minh Quan Dang, and Kostas Pentikousis. Energy-efficient cloud computing. *The computer journal*, 53(7):1045–1051, 2010.

[41] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In Alistair Barros, Daniela Grigori, Nanjangud C. Narendra, and Hoa Khanh

Dam, editors, *Service-Oriented Computing*, pages 316–323, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48616-0.

[42] IBM. Cplex optimizer. URL https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer. Last visited: 02.06.18.

[43] Poul Lumholtz. Fpga briefing part ii fpga development. URL www.teknologisk.dk/_root/media/24224_FPGA_briefing_part_2.pdf. Last visited: 13.05.18.

[44] Docker. What is docker, . URL https://www.docker.com/what-docker. Last visited: 09.04.18.

[45] Docker. Get started, part 1: Orientation and setup, . URL https://docs.docker.com/get-started/#docker-concepts. Last visited: 13.04.18.

[46] Docker. Swarm mode overview, . URL https://docs.docker.com/engine/swarm/. Last visited: 04.06.18.

[47] Docker. Get started, part 2: Containers, . URL https://docs.docker.com/get-started/part2/. Last visited: 13.04.18.

[48] Docker hub. URL https://hub.docker.com/.

[49] Docker. Dockerfile reference, . URL https://docs.docker.com/engine/reference/builder/. Last visited: 04.04.18.

[50] Docker. Runtime metrics, . URL https://docs.docker.com/config/containers/runmetrics/#docker-stats. Last visited: 03.05.18.

[51] Digital Ocean. Digital ocean. URL https://www.digitalocean.com/. Last visited: 11.04.18.

[52] Ian Fette. The websocket protocol. 2011.

[53] Amazon Web Services. Amazon ec2 f1 instances, September 2017. URL https://aws.amazon.com/ec2/instance-types/f1/.

[54] Amazon Web Services. Amazon ec2 instance types, . URL https://aws.amazon.com/ec2/instance-types/. Last visited: 02.05.18.

[55] Gibibyte. URL https://no.wikipedia.org/wiki/Gibibyte. Last visited: 31.05.18.

[56] Amazon Web Services. Amazon ec2 pricing, . URL https://aws.amazon.com/ec2/pricing/on-demand/. Last visited: 02.05.18.

[57] Amazon Web Services. Fpga developer ami, . URL https://aws.amazon.com/marketplace/pp/B06VVYBLZZ. Last visited: 01.05.18.

[58] Amazon Web Services. aws-fpga, September 2017. URL https://github.com/aws/aws-fpga.

[59] Amazon Web Services. Use opencl development environment with amazon ec2 f1 fpga instances to accelerate your c/c++ applications, also f1 instances are now available in us west (oregon) and eu (ireland) regions, . URL https://aws.amazon.com/about-aws/whats-new/2017/09/use-opencl-development-environment-with-amazon-ec2-f1-fpga-instances-to-accelerate-your-c-c-plus-plus-applications-also-f1-instances-are-now-available-in-us-west-oregon-and-eu-ireland-regions/. Last visited: 01.05.18.

[60] Zebra on 1 fpga (image classification). URL https://aws.amazon.com/marketplace/pp/B0719156K8?qid=1525272059272&sr=0-2&ref_=srh_res_product_title. Last visited: 02.05.18.

[61] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[62] University of California Berkeley. Berkeley logic interchange format (blif). https://www.cse.iitb.ac.in/~supratik/courses/cs226/spr16/blif.pdf, July 1992.

[63] Cotton Seed. Arachne-pnr, 2015. URL https://github.com/cseed/arachne-pnr. Last visited: 20.02.18.

[64] Odin ii. URL http://docs.verilogtorouting.org/en/latest/odin/. Last visited: 14.05.18.

[65] Icarus verilog. URL http://iverilog.icarus.com/. Last visited: 14.05.18.

[66] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[67] Ubuntu 16.04.4 lts (xenial xerus). URL http://releases.ubuntu.com/16.04/. Last visited: 01.05.18.

[68] Virtualbox. URL https://www.virtualbox.org/. Last visited: 01.05.18.

[69] Docker. Get docker ce for ubuntu, . URL https://docs.docker.com/install/linux/docker-ce/ubuntu/. Last visited: 01.05.18.

[70] Gnu make. URL https://www.gnu.org/software/make/. Last visited: 21.05.18.

[71] Signalfx. docker-collectd. URL https://github.com/signalfx/docker-collectd. Last visited: 24.05.18.

[72] Ide8. URL https://staging.ide8.io/. Last visited: 04.06.18.

[73] Putty. URL https://www.putty.org/. Last visited: 21.05.18.

[74] Xming. URL https://sourceforge.net/projects/xming/. Last visited: 21.05.18.