FPGA-Augmented Intelligent Devices for the Internet of Things

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Ingenieurswissenschaften

genehmigte Dissertation

von

Alwyn Johannes Burger

aus

Mossel Bay, Südafrika

1. Gutachter: Univ-Prof. Gregor Schiele
2. Gutachter: Prof. Dr.-Ing. habil. Sven Tomforde

Tag der Mündlichen Prüfung: 23.02.2023

Acknowledgements

I would like to express my deepest gratitude to my supervisor Gregor Schiele, without whom I would definitely not have been able to complete this journey. Your guidance and assistance has been invaluable, and you have been a thoroughly positive influence on my life. I am also incredibly grateful to my colleagues at the Embedded Systems group for their companionship throughout our many years together. To my various co-authors I also extend my thanks for assisting me in this work.

Lastly, I would like to extend my sincere thanks to Wiebke for supporting me every day, my family for everything they have done for me, and my friends for providing much-needed distraction.

Abstract

The Internet of Things (IoT) is constantly expanding with more complex applications being developed. This depends on increased artificial intelligence (AI) both to compute the application logic and manage the vast numbers of devices. To avoid being dependent on a constant connection to the cloud, groups of local devices can be augmented with additional computational abilities through hardware acceleration using e.g. field programmable gate arrays (FPGAs). Allowing these devices to cooperate by offloading tasks to each other should create highly adaptive and efficient deployments.

This work introduces the Elastic Node, an FPGA-augmented smart IoT device focussed on increasing local AI on the device. It offers a hardware platform to assist experimentation with local hardware acceleration in IoT deployments, as well as a runtime in the form of a middleware that facilitates software development through a stub-skeleton abstraction. This provides an easy-to-use runtime that functions without a local OS, offering convenient embedded hardware acceleration on a platform that operates under 200 mW and can switch between arbitrary accelerators within 100 ms.

Due to the limited configurable resources available on embedded FPGAs, design optimisations are required to allow more sophisticated AI to be deployed. Complex machine learning algorithms such as convolutional neural networks (CNNs) and incremental principle component analysis (IPCA) can be computed directly on the embedded IoT device. The presented novel IPCA hardware accelerator design reduces DSP slice resource usage by 84% when compared to conventional CORDIC-based implementations.

Since manually optimising dynamic and adaptive systems like these is not feasible, a reinforcement learning approach is used to create a local device agent. It optimises the device behaviour by addressing the augmented offloading problem that considers local hardware acceleration and peer-to-peer offloading. Using Q-learning and shallow neural networks, various agents are developed that perform up to 150% more jobs than an agent acting randomly.

Kurzfassung

Das Internet der Dinge (englisch, Internet of Things, IoT) gewinnt immer mehr an Bedeutung. Immer komplexere Anwendungen und Systeme werden entwickelt. Hierzu werden häufig Verfahren der Künstlichen Intelligenz (KI) eingesetzt, sowohl um die eigentliche Anwendungslogik zu berechnen als auch zur Verwaltung der stetig wachsenden Gerätepopulation. Um unabhängig von einer durchgängigen Verbindung zu einer entfernten Rechencloud zu sein, können lokale Gerätegruppen mit weitgehenden Datenverarbeitungsfähigkeiten ausgestattet werden, z.B. durch Integration von Hardwarebeschleunigern auf Basis von Field Programmable Gate Arrays (FPGAs). Die lokale Kooperation solcher erweiterter Geräte erlaubt es – beispielsweise durch dynamische Auslagerung von Berechnungen (sog. Offloading) – hochadaptive und effiziente Systeme zu entwickeln, die sich zur Laufzeit an Veränderungen in ihrer Ausführungsumgebung anpassen.

In dieser Arbeit wird untersucht, wie solche, mit FPGAs erweiterte intelligente IoT-Geräte, entworfen und eingesetzt werden können, sodass lokale KI-Fähigkeiten verbessert werden können. Hierzu wird mit dem Elastic Node eine Hardwareplattform vorgestellt, die es erlaubt, Experimente und Prototypen zu entwickeln um das Potential lokaler Hardwarebeschleunigung für IoT-Systeme zu untersuchen. Diese Hardware wird durch eine Software-Abstraktion auf Basis lokaler Stellvertreter ergänzt, die eine einfach zu verwendende, betriebssystemunabhängige Ausführungsumgebung anbietet, die für sehr ressourcenarme eingebettete Systeme optimiert ist. Das resultierende System hat eine Leistungsaufnahme von unter 200 mW und kann dynamisch innerhalb von 100 ms zwischen verschiedenen Hardwarebeschleunigern wechseln.

Aufgrund der stark eingeschränkten Ressourcen eingebetteter FPGAs werden zudem Optimierungstechniken für den Entwurf eingebetteter KI untersucht und eine Reihe von Hardwarebeschleunigern für komplexe maschinelle Lernverfahren entwickelt. Damit können z.B. Convolutional Neural Networks (CNNs) und Incremental Principle Component Analysis (IPCA) direkt auf dem eingebetteten IoT-Gerät ausgeführt werden. Der vorgestellte neuartige IPCA-Beschleuniger kann so beispielsweise die notwendigen digitalen Signalprozessoren um 84% reduzieren im Vergleich zur CORDIC-basierten Referenzimplementierung, und läuft deutlich schneller als Lösungen für die Cloud.

Da eine solche manuelle Optimierung auf Systemebene nicht sinnvoll ist, wird schließlich untersucht, wie ein in lokalen Gerätegruppen ausgeführtes Reinforcement Learning mit lokalen Geräteagenten dazu verwendet werden kann, das Systemverhalten dynamisch zu adaptieren und zu optimieren. Hierzu wird eine verteilte Lösung für das sogenannte erweiterte Offloading-Problem entwickelt und evaluiert, die Berechnungen sowohl auf IoT-Geräte in der selben Umgebung, aber auch auf den lokalen FPGA auslagern kann. Durch Verwendung von Q-Learning und flachen neuronalen Netzen können verschiedene Agententypen entwickelt werden und eine um 150% höhere Anzahl von Rechenaufgaben im Vergleich zu einer randomisierten Lösung bearbeitet werden.

Contents

Acronyms **xi**

1 Introduction **1**
1.1 Research Hypothesis 4
1.2 Research Questions 5
1.3 Scientific Contributions 6
1.4 List of Publications 6
1.5 List of Supervised Theses 7
1.6 Outline 8

2 System Overview **11**
2.1 Use Cases 11
2.1.1 Autonomous Drones 11
2.1.2 IoT System 14
2.2 System Model 16
2.3 Typical Implementation 18
2.4 System Requirements 20

3 Fundamentals **23**
3.1 Accelerating Computation in Embedded Systems 23
3.1.1 Multicore and Multithreaded Systems 24
3.1.2 Field Programmable Gate Arrays (FPGAs) 25
3.1.3 Graphics Processing Units (GPUs) 26
3.1.4 Heterogeneous Computing 26
3.2 Optimising Hardware Acceleration Performance 27
3.2.1 Pipelining 27
3.2.2 Parallelism 28
3.2.3 Batching 28
3.3 Heterogeneous Application Models 30
3.3.1 OS Threads 30
3.3.2 Modular Tile 31
3.3.3 System on Chip 31
3.3.4 Middleware 32

3.4 Connected Computing Paradigms . . . 32
3.4.1 Offloading . . . 32
3.4.2 Placement . . . 34
3.4.3 Distributed Computing . . . 34
3.4.4 Edge Computing . . . 34
3.4.5 Mobile Edge Computing . . . 35
3.4.6 Fog Computing . . . 35
3.4.7 Peer-to-peer and Grid Computing . . . 35
3.5 Intelligent Devices . . . 35
3.5.1 Self-x Computing . . . 36
3.5.2 Organic Computing . . . 37
3.5.3 Autonomic Computing . . . 38
3.5.4 Machine Learning Techniques . . . 38

4 Elastic Node Platform 47
4.1 Motivation and Background . . . 47
4.2 Platform Requirements . . . 48
4.3 Platform Overview . . . 50
4.3.1 Hardware Functions . . . 51
4.3.2 Middleware . . . 52
4.3.3 Hardware-as-a-Service . . . 54
4.4 Stub-Skeleton Abstractions . . . 55
4.4.1 Interface Description Language . . . 56
4.4.2 Skeleton Interface Definition . . . 57
4.4.3 Transparent I/O Caching . . . 60
4.4.4 Microcontroller Unit (MCU)-Field Programmable Gate Array (FPGA) Offloading Procedure . . . 61
4.5 Hardware Platform Design . . . 62
4.5.1 Hardware Interconnect . . . 63
4.5.2 Power Monitoring . . . 65
4.6 Discussion . . . 67

5 Optimising Embedded AI Accelerator Design 69
5.1 Optimisation Approach . . . 69
5.2 Hardware Architecture Optimisations . . . 70
5.2.1 DSP Timing Optimisations . . . 72
5.2.2 Volatile and Non-Volatile Memory Tiers . . . 72
5.2.3 Minimising Expensive Operations . . . 73
5.2.4 Floating Point Representation . . . 73
5.2.5 Utilising LUTs for Precomputation . . . 74
5.2.6 Latency and Throughput Modelling . . . 74
5.3 Example Hardware Accelerators . . . 75

5.3.1 Artificial Neural Networks 75
5.3.2 Convolutional Neural Networks 77
5.3.3 IPCA 82
5.3.4 Summary 89

6 Learning Intelligent Devices 91
6.1 Problem Statement 92
6.2 Design Rationale 93
6.3 Requirements 94
6.4 Analytical Model 95
6.5 Agent Design 98
6.5.1 State-Action Decisions 99
6.5.2 System State 102
6.5.3 Action Space 103
6.5.4 Reward Functions 105

7 Evaluations 109
7.1 Phase 1: Elastic Node Viability 110
7.1.1 Energy Consumption 111
7.1.2 Accelerator Switching Latency 114
7.1.3 Elastic Node Middleware Resource Overhead 116
7.1.4 Development Complexity 118
7.2 Phase 2: Hardware Accelerator Optimisation 123
7.2.1 Optimal Fixed Point Representation 124
7.2.2 Common Hardware Design Techniques 126
7.2.3 CNN Latency Model Verification 131
7.2.4 Incremental PCA Facial Detection Use Case 132
7.3 Phase 3: Intelligent Cooperating Devices 133
7.3.1 Experimental Setup 133
7.3.2 Agent Comparison 135
7.3.3 System State Limitations 137
7.3.4 Catastrophic Failure 140
7.3.5 Heterogeneous Agents 143
7.3.6 Comparing Q-table and SRL Agents 145
7.3.7 Q-table vs SRL Agents in Dynamic Environments 147
7.4 Concluding Thoughts 149

8 Related Work 151
8.1 Platform Related Work 152
8.1.1 Heterogeneous Embedded Platforms 152
8.1.2 General Distributed Heterogeneous Middleware 154
8.1.3 Offloading Mechanism 155

8.1.4 Distributed Embedded Middleware . . . 156
8.1.5 FPGA-Based Middleware . . . 157
8.1.6 Self-Aware Distributed Systems . . . 158
8.1.7 System Modelling for Distributed Systems . . . 158
8.1.8 self-x Reconfigurable Systems . . . 159
8.2 Intelligent Embedded Offloading . . . 159
8.2.1 Greedy and Short-Term Learning Approaches . . . 160
8.2.2 Reinforcement and Deep Learning approaches . . . 161

9 Conclusion and Outlook 165
9.1 Contributions . . . 165
9.2 Research Questions . . . 167
9.3 Outlook . . . 167

A Interface Description Language Specification 171
A.1 Grammar . . . 171
A.2 Fields . . . 174
A.2.1 MCU Configuration . . . 174
A.2.2 Function Configuration . . . 175
A.3 Configuration Mapping . . . 176
A.4 Types . . . 177
A.5 Defaults . . . 178

B ANN Interfaces 181
B.1 ANN VHDL Entity . . . 181

C ANN Skeleton 183
C.1 ANN Stub . . . 185

D Evolution of Hardware Versions 189
D.1 Elastic Node v2 . . . 189
D.2 Elastic Node v3 . . . 190
D.3 Elastic Node v4 . . . 191
D.4 ARM Elastic Node . . . 192
D.5 ARM Elastic Node v2 . . . 193

Bibliography 195

Acronyms

A2C	Advantage Actor Critic. 41
A3C	Asynchronous Advantage Actor Critic. 41
AC	Actor Critic. 41
AI	Artificial Intelligence. 1–4, 6, 8, 11, 12, 14, 18–20, 22, 23, 29, 38, 47, 52, 69, 75, 77, 80, 83, 90, 93, 109, 133, 148–151, 165–168
ANN	Artificial Neural Network. x, 42, 44, 75–77, 80, 81, 89, 111, 113, 118, 119, 121, 123, 165, 180, 181, 190
API	Application Programming Interface. 32, 51, 65, 67, 110, 118, 119, 121, 122, 181, 185
ASIC	Application Specific Integrated Circuit. 25, 49
AVX	Advanced Vector Extensions. 24
AXI	Advanced eXtensible Interface. 27, 48, 52, 58, 157
BGA	Ball Grid Array. 191
BNN	Binary Neural Network. 80
BRAM	Block RAM. 25, 72, 73, 76, 80, 112–115
CLB	Configurable Logic Block. 25
CNN	Convolutional Neural Network. 6, 42, 44, 52, 73, 75, 77–81, 89, 112, 123–125, 131, 165, 166, 168, 191
CORDIC	COrdinate Rotation DIgital Computer. 84, 128, 166
COTS	Commercial off the Shelf. 27, 48, 50, 62, 67, 151, 167
CPLD	Complex Programmable Logic Device. 152
CPU	Central Processing Unit. 4, 24–27, 44, 73, 76, 79, 130, 153, 157, 160, 161
CV	Computer Vision. 2–4, 11, 12, 20, 28, 59
DDR	Double Data Rate. 76
DL	Deep Learning. 45, 161
DMA	Direct Memory Access. 157
DNN	Deep Neural Network. 42–44, 162
DOA	Distributed Object API. 154
DOL	Division of Labour. 140–143

DPR Dynamic Partial Reconfiguration. 157
DQN Deep Q Network. 44, 161
DRAM Dynamic Random Access Memory. 76, 112
DRL Deep Reinforcement Learning. 41, 45
DSP Digital Signal Processing. 15, 24, 25, 70, 72, 74, 80, 86, 117, 118, 124–128, 166
DWT Discrete Wavelet Transform. 80

EBNF Extended Backus-Naur Form. 119, 171
ECC Error Correction Code. 159
ECG Electrocardiogram. 77–81
EDK Embedded Development Kit. 19
EH Energy Harvesting. 69, 160
ELM Extreme Learning Machine. 161
ETSI European Telecommunications Standard Institute. 35

FAAS Fully Autonomous Aerial System. 132
FC Fully Connected. 42, 44, 81
FIFO First In First Out. 155
FIR Finite Impulse Response. 60, 71, 74, 80, 112
FPGA Field Programmable Gate Array. viii, 3, 4, 6, 8, 13, 14, 16–19, 21–23, 25–27, 29, 30, 43, 47–65, 68–76, 79, 80, 83, 84, 88–92, 96, 97, 102, 109–116, 118, 119, 122–131, 133, 135, 141, 149, 150, 152–157, 165–169, 171, 177, 179, 189–192
FPS Frames Per Second. 132

GPU Graphics Processing Unit. 4, 25–27, 29, 44, 73, 151, 157
GUI Graphical User Interface. 56, 58

HAL Hardware Abstraction Layer. 166
HDL Hardware Description Language. 18, 25, 26, 52, 53, 56, 74, 81
HLS High-Level Synthesis. 19
HPC High-Performance Computing. 35, 157
HWF Hardware Function. 51–57, 59–62, 70, 76, 78, 79, 84, 86, 88, 111, 114–121, 123, 124, 126, 129, 131, 132, 150, 166, 171, 176–181, 185, 188

I^2C Inter-Integrated Circuit. 66

ICAP	Internal Configuration Access Port. 115, 116
IDL	Interface Description Language. 9, 56–58, 60, 77, 118, 119, 121, 122, 166, 171, 174, 177–183, 188
IoT	Internet of Things. 1–6, 11, 14, 15, 20–23, 35, 47, 48, 68, 75, 77, 91, 92, 109, 110, 123, 149, 151, 155–157, 160, 165–167, 169
IP	Internet Protocol. 154
IPCA	Incremental Principle Component Analysis. 6, 75, 82–84, 86–89, 123, 126, 129, 131, 132, 149, 165, 166, 168
ISR	Interrupt Service Routine. 121
JTAG	Joint Test Action Group. 54, 123
LCS	Learning Classifier System. 41, 159
LIDAR	Light Imaging, Detection and Ranging. 143
LSB	Least Significant Bit. 74
LUT	Lookup Table. 25, 72, 74, 80, 81, 87, 88, 112–116, 118, 126
MAC	Multiply-ACcumulate. 25, 72, 74, 124
MAPE-K	Monitor-Analyse-Plan-Execute over a shared Knowledge. 38
MCC	Mobile Cloud Computing. 33, 160, 161, 163
MCU	Microcontroller Unit. viii, 4, 6, 16, 21, 24, 47–57, 59–68, 81, 90, 96, 97, 111, 115, 116, 118, 120, 121, 123, 152–157, 166–168, 171, 174, 176, 177, 179, 189, 190, 192, 193
MDP	Markov Decision Process. 40, 160
MEC	Mobile Edge Computing. 33, 35, 160, 161, 163
ML	Machine Learning. 4, 20, 38, 52, 75, 90, 93, 165, 167
MLP	Multilayer Perceptron. 42, 75
MNIST	Modified National Institute of Standards and Technology. 161
MOM	Message-Oriented Middleware. 122
MUX	Multiplexer. 74
NLP	Natural Language Processing. 2, 4, 15, 77
ODE	Ordinary Differential Equation. 160
OS	Operating System. 19, 24, 30–32, 52, 55, 153–158, 168

OTA	Over-The-Air. 16, 53, 115, 156
P2P	Peer-to-Peer. 35, 91, 158, 159
PC	Personal Computer. 4, 153, 154, 157
PCA	Principle Component Analaysis. 52, 82, 83, 128
PCB	Printed Circuit Board. 32, 189
PCIe	Peripheral Component Interconnect Express. 48, 152, 153
PL	Programmable Logic. 27, 68, 80, 113, 126, 166, 190
POI	Point of Interest. 59
QoS	Quality of Service. 156
RAM	Random Accessed Memory. 25
RAN	Radio Access Network. 161
REST	Representational State Transfer. 55, 154, 155
RF	Radio Frequency. 153
RISC	Reduced Instruction Set Computer. 168
RL	Reinforcement Learning. 39–41, 45, 55, 93, 149, 161, 165–167
RNN	Residual Neural Network. 44
RPC	Remote Procedure Call. 32, 34, 55, 59, 61, 68, 154, 157, 166
RRF	Remote Resource Framework. 55
RRH	Remote Radio Head. 41
RTC	Real-Time Clock. 67, 169
RTI	Runtime Infrastructure. 155
RTOS	Real-Time Operating System. 19, 24, 30, 52, 168
SaaS	Software as a Service. 54
SBC	Single-Board Computer. 34
SDR	Semidefinite Relaxation. 160
SDRAM	Synchronous Dynamic RAM. 72
SGR	Squared Givens Rotation. 83–88
SHARC	Super Harvard Architecture Single-Chip Computer. 24
SIMD	Single Instruction Multiple Data. 24, 26, 29, 44
SLA	Service Level Agreement. 156
SNN	Shallow Neural Network. 43, 101, 106, 145, 162
SNR	Signal-to-Noise Ratio. 80
SoC	System on Chip. 4, 24, 27, 30–32, 55, 58, 68, 70, 122, 152, 153, 157, 159

SPI	Serial Peripheral Interface. 53, 64, 116
SRAM	Static Random Access Memory. 53, 76, 97, 112, 153
SRL	Shallow Reinforcement Learning. 101, 102, 133, 145–149, 161, 166
SUNN	Small Unorganized Neural Networks. 161
TQFP	Thin Quad Flat Pack. 191
UART	Universal Asynchronous Receiver/Transmitter. 64
UAV	Unmanned Aerial Vehicle. 11, 12, 83, 89, 123, 132, 133, 156
USB	Universal Serial Bus. 63, 65, 66, 116, 191
VHDL	Very High Speed Integrated Circuit Hardware Description Language. x, 18, 25, 56–58, 75, 119, 175, 177, 181
VM	Virtual Machine. 154, 156
VSN	Visual Sensor Network. 156, 158
WSN	Wireless Sensor Network. 111
XCS	eXtended Classifier System. 41
XMEM	External Memory. 64–66, 186

Chapter 1.

Introduction

As the Internet of Things (IoT) [203] becomes more ubiquitous in our daily lives, the number of computations required by its applications grows. Every smart light or thermostat creates additional processing that needs to be done. More complex IoT products such as smart vision (e.g. doorbell cameras) has increased the intelligence required on the device. Instead of being the basic data input/output system they were originally envisioned as, they suddenly include highly complex processing. Consider for example the smart vacuum, which took a basic cleaning appliance and augmented it with the ability to map its environment, plan its routine, and use Artificial Intelligence (AI) for decision-making. This allows it to navigate to specific locations, understand voice commands, and even learn its owner's habits to schedule cleaning routines.

When combined with the common dependency on mobile and battery-operated devices (e.g. wearables), these systems require a fine balance between processing power and energy efficiency. Instead of trying to deploy processing powerhouses for every local device, the IoT adopted the popular cloud-centric system stack [203].

Within this hierarchy, most of the computational complexity is placed in the cloud, while local devices remained simple in order to minimise their price and power consumption. For example, consider the use case shown in Figure 1.1 where Fred has a smart home. His smart bulbs typically do not perform any processing at all, accepting wireless messages to either turn on and off or report their current state. To connect them to the internet as a whole, Fred's WiFi router acts as a gateway, and all of the AI and application complexity involved in automating his bulbs is deferred to the cloud services of the bulbs' manufacturer. This includes basic rules such as switching his lights on when he gets home, or more complex tasks like learning his daily routine to pre-emt his needs.

Regardless, his "smart" bulbs have very low intelligence, simply responding to control messages from the cloud. These deployed embedded devices form the bottom level of a typical IoT hierarchy as shown in Figure 1.2. The cloud is at the top, while the middle layer (generally referred to as a gateway[1]) is mostly responsible for networking, allowing the local devices to connect to the cloud services via the internet.

However, now that Fred is introducing more complex devices like voice controlled hubs

[1] `https://internetofthingsagenda.techtarget.com/definition/IoT-gateway` (last visited: 2021-12-13)

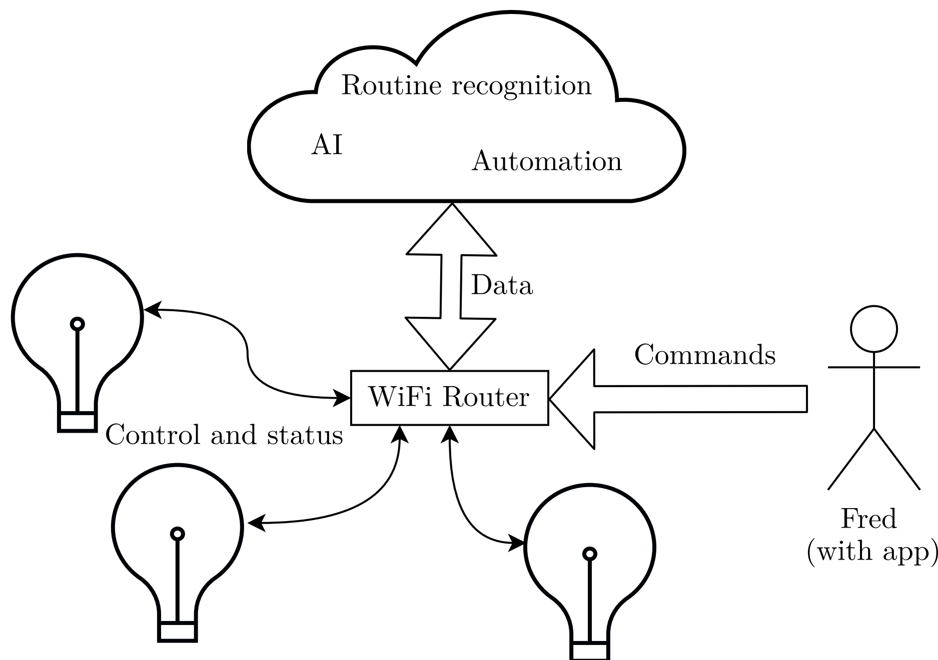


Figure 1.1.: Example use case with Fred controlling his smart bulbs through the IoT, relying on the cloud to provide the intelligence required and his WiFi router to bridge the connection between his bulbs, his app and the cloud.

into his home, recordings of his voice are also being processed using Natural Language Processing (NLP), and his door camera is being checked for intruders using Computer Vision (CV). While delegating all of this complexity to the cloud simplifies the local devices, it creates a number of concerns: someone could listen in on his private conversations, or burglars could use his own door camera to see when he is not home.

This highlights a big issue with having all of the AI in the cloud: complex tasks such as learning user habits require the collection of a lot of personal data. This comes with privacy concerns, as the customer inherently relies on cloud services' ability to protect their private images, videos, and audio recordings – instead of the user being responsible for their own data locally. Sadly, countless examples can be found where people's privacy have been violated due to IoT cloud frameworks not protecting their customers' data properly (e.g. Peleton customer data breach[2], Xiaomi camera feed breach[3], and internet-connected fuel stations [99]).

Additionally, what happens when Fred's local internet connection is not stable? What if the bulb manufacturers do not maintain their hosted cloud services and they are

[2] `https://www.forbes.com/sites/emilsayegh/2021/07/22/peloton-breach-reveals-a-coming-iot-data-winter/` (last visited: 2021-10-09)

[3] `https://www.securitymagazine.com/articles/91502-xiaomi-mijia-camera-picking-up-strangers-camera-feeds` (last visited: 2021-12-13)

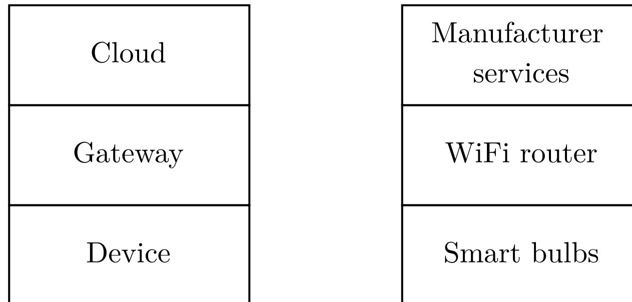


Figure 1.2.: Typical IoT system stack with the intelligence on top and low complexity at the bottom. The bulbs are simply controllable lights, while the cloud hosts complex AI (e.g. habit recognition). The WiFi router acts as connection gateway to provide access from the internet.

suddenly not available? What if intruders manage to interfere with his house's internet connectivity, effectively shutting his devices off from the outside world? There is no way for his door camera to function properly without an internet connection when it depends on the system from Figure 1.2. This model is not well suited when reliability and low latency is required, since it depends on having a constant and reliable connection to the cloud service. Some examples include safety-critical devices like smoke detectors, or user-facing interfaces like motion sensors monitoring Fred's movement to pre-emptively turn on his lights. In these cases, it is important that the system offers a local-only operation mode that can continue to offer services during downtimes.

The way we see it, the conundrum facing such a system is the need to combine high energy efficiency, strong computational performance, and low cost. If sensitive data and the required intelligence is to be kept locally in the system, either the local embedded devices or the gateways need to become smarter so they can perform increased workloads.

Our goal is a more flexible solution that can adapt to an ever-changing role in the system. When only a basic task such as occasional sensing is being performed, it should maximise energy efficiency. On the other hand, when a more computationally complex task such as CV or AI is required it should temporarily 'grow' to handle it. This type of elastic system should provide the flexibility of always having the correct device for the task, without compromising performance or power efficiency.

Numerous approaches are available for this type of flexibility, depending on where the additional processing capabilities are deployed. At the core of this is the *offloading problem*, where devices either locally compute tasks or offload them somewhere else. While remote options exist that send data and computation to a data centre or server, this suffers from the cloud dependency discussed above. Instead, we target a local solution that augments an embedded device with enough processing power to satisfy the smart IoT without sacrificing the efficiency that small and mobile devices require.

One very interesting option for local hardware acceleration is an FPGA that allows the

deployment of semi-arbitrary hardware architectures. By *reconfiguring* at runtime, they can adapt to changing system roles or even different applications. Within a fraction of a second, they can instantiate the ideal hardware architecture for computing practically any problem. They are particularly well-suited to acting as coprocessors, creating a local heterogeneous systems alongside a sequential processing unit such as an MCU. However, they can also be used as standalone System on Chips (SoCs) by instantiating a so-called soft processing core to perform the general processing tasks. Either way, having such a combination of sequential and accelerated processing creates tremendous opportunities for adaptive embedded systems.

Traditionally, FPGAs have seen relatively low adoption and popularity in the embedded space when compared to general purpose solutions such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs). While practically every mobile phone and Personal Computer (PC) includes both of these components, FPGAs have mostly been relegated to larger industrial usage such as enterprise management [156, 276] or research.

However, recently a noticeable increase in commercial FPGA adoption has been seen, with a number of products even utilising FPGAs without the public being aware of it. Some examples include the NVidia G-Sync HDR module in high-end computer monitors [1], or the discovery during a teardown that the iPhone 7 included an FPGA for unknown purposes [226].

1.1. Research Hypothesis

Efficiently utilising local hardware acceleration in smart IoT devices has the potential to create powerful AI-capable distributed systems. Without introducing latency and energy costs from transferring large amounts of data, they can bring greater intelligence to IoT applications. This leads to more complicated systems that can be very difficult to optimise, due to the vast solution space introduced by adding reconfigurable FPGAs to each device. Especially when considering numerous devices operating in full cooperation, this leads to a huge optimisation problem for governing the system behaviour. Ideally, each device should be able to regulate its own behaviour to improve scaling when creating systems with large numbers of devices.

Therefore, our hypothesis is:

Connected and autonomic FPGA-augmented smart IoT devices can use AI to optimise their behaviour.

We identify the need to support two different types of AI in such a system. Along with the *application AI* required to execute the application itself (e.g. CV or NLP), some *device AI* is needed to let each device operate rationally [206]. Adding this sophistication through Machine Learning (ML) provides the advantage that the device can learn how

to behave without painstaking manual optimisation by a developer. The goal is to have *self-aware* devices that are aware of their own state as well as that of their environments – within the power limitations of a small embedded device. This should create a system capable of massive scale, since the devices can optimise locally without having to consider the global system.

1.2. Research Questions

Considerable work has been done in incorporating hardware acceleration and reconfigurable computing into embedded systems. Additionally, addressing the offloading problem (and the accompanying intelligence it adds to a system through optimising device behaviour) has been approached by multiple different communities. The novelty in our approach comes from incorporating the intricacies of using hardware acceleration on heterogeneous embedded systems with intelligently offloading distributed systems. To the best of our knowledge, ours is the first system that aims to solve the combination of these two problems.

Approaching our hypothesis requires investigation into two separate but related core research questions:

- **RQ1:** How can adaptive hardware acceleration increase the local intelligence of IoT devices?
- **RQ2:** How can distributed heterogeneous embedded devices learn to autonomously achieve shared goals?

These two questions highlight the multi-phase nature of this work. The first phase involves having an appropriate hardware/software runtime to perform our investigation. RQ1 and RQ2 cover the main research objectives of the second and third phases in turn: local intelligence through optimised hardware accelerators and distributed autonomous learning. These three phases will form the foundation of our approach, and upon their completion we will have either validated or invalidated our hypothesis.

As both of these research questions rest on assumptions of possibility (e.g. the ability of adaptive hardware acceleration to increase local intelligence) they will be answered by proving *whether* the relevant assumption is true, and by describing *how* each is possible. For RQ1 this focusses on surpassing the intelligence possible on IoT devices through traditional designs (i.e. without hardware acceleration), which requires a detailed definition of what is meant by *intelligence* as it is a very loaded term used to describe various characteristics.

RQ2 attempts to accomplish something very specific in the devices' ability to learn. It requires a detailed study of what is currently possible in the literature (within our definition of heterogeneity), and improving the performance (e.g. goal-achieving ability or robustness) of the cooperating devices.

1.3. Scientific Contributions

Through the three phases of our approach and by investigating our hypothesis, we made a number of contributions to the relevant scientific communities. The most important of these contributions include:

1. We designed and built a novel hardware platform for FPGA-augmented connected IoT devices – incorporating fine-grained energy monitoring and shared application logic between the MCU, FPGA and remote offloading,

2. We showed how the development complexity of embedded heterogeneous applications can be reduced using a familiar stub-skeleton abstraction, and provide a software runtime that includes hardware interface drivers,

3. We presented various advanced AI hardware accelerations and showed how they can efficiently utilise the available resources of embedded reconfigurable hardware using accelerator design techniques, and

4. We developed a device-local AI algorithm that optimises device behaviour when acting in cooperating teams, extending on the offloading problem to fully utilise FPGA-based hardware acceleration and emergent collaboration between peers.

These contributions were all peer reviewed and published through a combination of a poster [32], a demonstration [31], a workshop paper [35], full papers presented at conferences [34, 36, 211], and two published journal articles [33, 37]. A number of bachelor's and master's theses were also supervised during the course of this work. A list of these publications and theses can be seen in Sections 1.4 and 1.5 respectively, provided as proof of the community's acceptance of this work.

1.4. List of Publications

We provide here a complete list of the published scientific works associated with this dissertation. The first four entries relate to the Elastic Node as presented in Chapter 4, the next two to the Convolutional Neural Network (CNN) and Incremental Principle Component Analysis (IPCA) accelerators described in Chapter 5, and the last two to the goal-based agents from Chapter 6. Among the list is one workshop paper (PerIoT 2020), three full papers (ICAC 2019, ARCS 2020, and ACSOS 2020), and two journal articles (FGCS and TAAS).

1. Alwyn Burger, Christopher Cichiwskyj, and Gregor Schiele. *Elastic Nodes for the Internet of Things: A Middleware-Based Approach.* International Conference on Autonomic Computing (ICAC), pages 73–74. IEEE, 2017. ISBN 978-1-5386-1762-5. doi: 10.1109/ICAC.2017.27

2. Alwyn Burger and Gregor Schiele. *Demo Abstract: Deep Learning on an Elastic Node for the Internet of Things*. International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pages 555–557. IEEE, 2018. ISBN 9781538632277. doi: 10.1109/PERCOMW.2018.8480160

3. Gregor Schiele, Alwyn Burger, and Christopher Cichiwskyj. *The Elastic Node: An Experimentation Platform for Hardware Accelerator Research in the Internet of Things*. International Conference on Autonomic Computing (ICAC), pages 84–94. IEEE, 2019. ISBN 9781728124117. doi: 10.1109/ICAC.2019.00020

4. Alwyn Burger, Christopher Cichiwskyj, Stephan Schmeißer, and Gregor Schiele. *The Elastic Internet of Things - A Platform for Self-Integrating and Self-Adaptive IoT-Systems with Support for Embedded Adaptive Hardware*. Future Generation Computer Systems, 113:607–619, 2020. doi: 10.1016/j.future.2020.07.035

5. Alwyn Burger, Patrick Urban, Jayson Boubin, and Gregor Schiele. *An Architecture for Solving the Eigenvalue Problem on Embedded FPGAs*. International Conference on Architecture of Computing Systems (ARCS), pages 32–43. Springer, 2020. doi: 10.1007/978-3-030-52794-5_3

6. Alwyn Burger, Chao Qian, Gregor Schiele, and Domenik Helms. *An Embedded CNN Implementation for On-Device ECG Analysis*. International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). IEEE, 2020. doi: 10.1109/PerComWorkshops48775.2020.9156260

7. Alwyn Burger, David King, and Gregor Schiele. *Reconfigurable embedded devices using reinforcement learning to develop action-policies*. International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, 2020. ISBN 9781728172774. doi: 10.1109/ACSOS49614.2020.00046

8. Alwyn Burger, Gregor Schiele, and David W King. *Developing Action Policies with Q-Learning and Shallow Neural Networks on Reconfigurable Embedded Devices*. ACM Transactions on Autonomous and Adaptive Systems, 15(4):1–25, 2021. doi: 10.1145/3487920

1.5. List of Supervised Theses

Ten separate bachelor's and master's degree theses were supervised through this work, a list of which is provided here. Loosely the Elastic Node platform relates to the first six theses on this list, while the optimised accelerator design relates more the last four. These are referred to within this dissertation as appropriate, describing how each of them are incorporated into this dissertation.

1. Sascha Christian Hevelke. *An Approach for Efficient Runtime Self-Configuration for Embedded Reconfigurable Platforms*. Master's thesis, University of Duisburg-Essen, 2016
2. Nicolas Frick. *FPGA Energy Monitoring and Control through Power State Optimization*. Bachelor's thesis, University of Duisburg-Essen, 2019
3. Anam Khalid. *Design and Implementation of an optimized SDRAM interface in Verilog*. Master's thesis, University of Duisburg-Essen, 2018
4. Miroslav Valov. *Over-the-air Updating of an Embedded Heterogeneous Platform Using 802.15.4*. Bachelor's thesis, University of Duisburg-Essen, 2019
5. Philip Schmidt. *Hardware-/Software-Codesign for an Embedded Energy Monitoring Daughterboard*. Bachelor's thesis, University of Duisburg-Essen, 2019
6. Philipp Winnekens. *Development of an Interface Description Language and Stub/Skeleton Generator for Embedded Heterogeneous Multicore Systems*. Master's thesis, University of Duisburg-Essen, 2020
7. Lingkon Hossain. *Analysis and Design of Audio Capturing Solutions for Low-Power Embedded Systems*. Master's thesis, University of Duisburg-Essen, 2019
8. Chao Qian. *Energy Efficiency Analysis and Optimisation of Convolutional Neural Networks in Embedded FPGAs*. Master's thesis, University of Duisburg-Essen, 2019
9. Patrick Urban. *Implementation of Distributed Computer Vision using Embedded FPGAs*. Master's thesis, University of Duisburg-Essen, 2020
10. Jinwen Du. *Noise Mitigation for CNN Classifiers in Embedded Environments*. Master's thesis, University of Duisburg-Essen, 2020

1.6. Outline

Before we can present the design of our solution, a better understanding of the desired system is gained in Chapter 2 by studying some use cases and thereby creating the system requirements. After this, some fundamental concepts are discussed in Chapter 3 to provide context for the technical work being discussed.

The technical contributions will be explained through the three aforementioned phases of the work. Initially, the design of our embedded heterogeneous runtime is presented in phase one in Chapter 4. This covers both the hardware platform as well as the software runtime provided for aiding development. This is followed by optimising the designs of hardware accelerations in Chapter 5 (phase two) that demonstrate some of the AI capabilities of the embedded FPGAs used. In phase three, additional device intelligence

is introduced that allows each device to learn how to behave rationally by solving the extended offloading problem in Chapter 6. A thorough evaluation of the system created in our work follows in Chapter 7, addressing whether the requirements we set for our system were satisfied.

After this, some alternative approaches from literature are discussed in Chapter 8 to provide context for our work. Lastly, some concluding thoughts and an outlook on possible further investigations are provided in Chapter 9. A number of Appendices have been provided at the end of this dissertation to clarify the Interface Description Language (IDL) developed, and to provide examples of how our system can be used.

Chapter 2.

System Overview

Our proposed system involves a number of components that need to be designed and implemented, ranging from an appropriate hardware platform to a learning algorithm. Firstly, however, an overview of the entire system is required to better define its structure and overall functionality.

Therefore, we start by investigating in Section 2.1 two use cases where we identified our system would be beneficial: smart autonomous drones and a smart home IoT system. This is followed by a discussion of a typical environment we are designing for in the system model in Section 2.2.

At this point the discussion will turn to our proposed solution, as we describe a typical implementation in Section 2.3. Lastly, we set out a number of requirements in Section 2.4 that we feel encompass what this system should achieve and provide.

2.1. Use Cases

It is important when designing a system to consider specific example scenarios and how that system could benefit them. In our case, the aim is to achieve a better understanding of what roles smart heterogeneous embedded devices could fulfil in each presented scenario. Of particular interest are how the devices are expected to *act*, what their *environment* should look like, and what they should *achieve*.

These examples represent applications with varying types of complexity – a group of drones solving a highly computationally complex problem in CV and AI, and an IoT system that needs to scale for larger and more dynamic systems. Our goal is then to use these two use cases to identify both a unifying system model and a set of requirements for our system.

2.1.1. Autonomous Drones

The first scenario to consider is a number of autonomous drones that collaborate to achieve a shared goal. One concrete example of this is the surveying of fields to establish crop yields [26], which involves the orchestrated imaging of a large area by a team of Unmanned Aerial Vehicles (UAVs). The drones autonomously decide where the next image should be taken, relocate to that position, and capture the image. This image

must then be processed using a combination of CV techniques and machine learning in order to calculate the crop yield for the particular grid in view. The last step then requires information to be collected from all the drones and combined to form an overall view of the area being surveyed, combining overlapped areas to improve accuracy.

This problem can be modelled as a mobile homogeneous swarm [59, 69, 115] where each device acts as a data source when it captures an image, but also as a processing source when it needs to be processed. Each of these roles are very homogeneous over time, since each image is processed largely in the same way. These problems are both computation and data heavy as the drones' cameras can capture large, detailed images. This highlights the benefit of processing them directly on the drones instead of transmitting the raw images to some edge or cloud service – reducing the time and energy cost of transmission which can be in the order of seconds for raw images [27].

The task graph for this is provided in Figure 2.1. It shows the four discrete stages of the processing pipeline, starting with the feature extraction (2). Here each image is searched for important features that can be used not only by the yield estimator (5), but also for path planning (4). This creates a non-sequential task graph, as the same extracted vectors should be used in multiple different processing steps. Once the images are converted to yield maps (3) using these features, the route planning component (4) can create the next waypoint for the drone. Since route planning requires information about all drones, it is usually implemented as a centralised component [26], causing a synchronisation requirement. This means that regardless of how the images are processed, the processed results need to be collected in one place – likely a server that can perform the complex final processing steps.

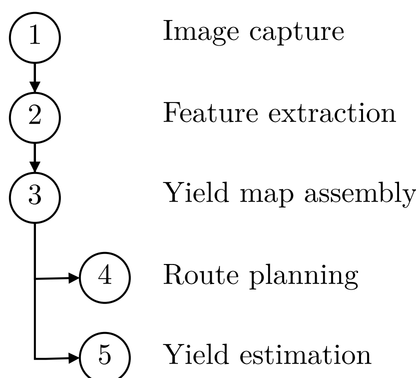


Figure 2.1.: Task graph for UAV use case

To map heterogeneous reconfigurable computing onto this problem, consider each of the steps in the task graph as an independent layer. At the top there is a sensing layer, followed by two CV layers, and the AI layer that performs the path planning. In our work we consider these processing layers to be *tasks* that need to be computed, either by the device itself or offloaded to an edge or cloud component.

Even a small task graph like this can create a large solution space for finding an optimal division of labour between the various drones and centralised computation. For example, consider the scenario in Figure 2.2 where all the drones capture an image and offload them to a server for processing. In this case, none of the drones need to do any processing locally, but multiple full images need to be sent through a long-distance communication interface (e.g. a 4G module) – which can be very energy intensive.

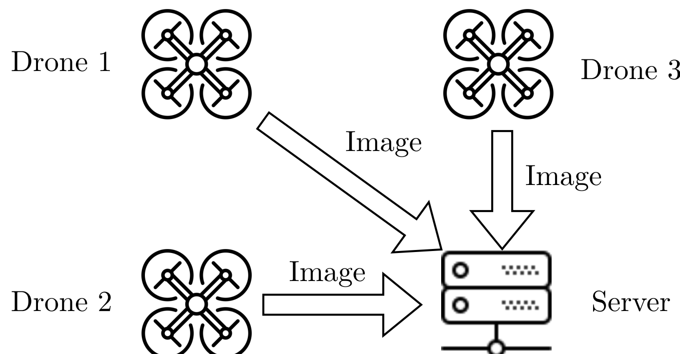


Figure 2.2.: Fully offloading scenario, where each drone captures an image and offloads them directly to server for processing.

Alternatively, each of the drones can locally process their captured images using their local FPGAs and only offload the processed results to the server – as shown in Figure 2.3. This means that they do not have to send the images at all, instead locally performing all the processing layers and only ever sending the extracted features and results. However, this requires each of the drones to power up and use their local FPGAs. This creates a very different solution to the first one, where the viability depends on the relative energy costs of offloading images and locally processing on the FPGA.

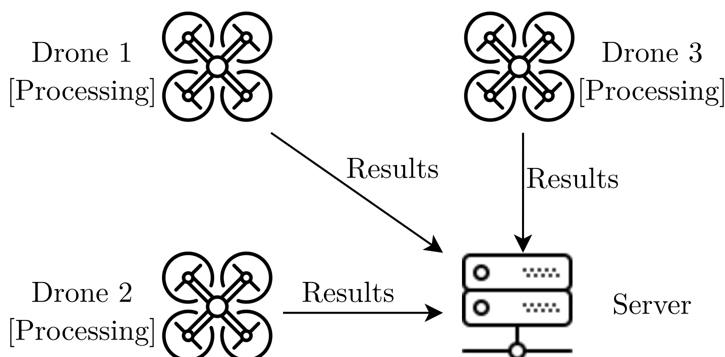


Figure 2.3.: Local processing scenario, where all three drones locally process their own images and only send the processed results to the server

Lastly, the drones can choose to offload their images to one processing drone, as shown

in Figure 2.4. This means that only a single drone needs to utilise its local FPGA, while the others use their local area communication (which should be more energy-efficient than the long-ranged one) to offload their image to the processing drone. This further expands the solution space, where different drones can take turns to be processing drone to balance their energy usage.

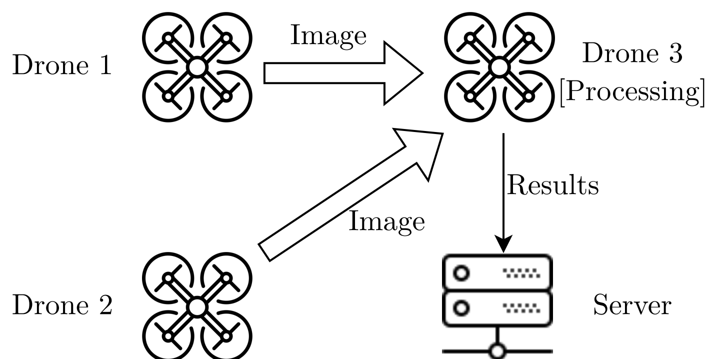


Figure 2.4.: Specialisation scenario, where a single drone is responsible for processing while the others offload their captured images

An important concept for this type of optimising in a cooperative system is *specialisation*, where each device focusses on a specific task instead of equally addressing all tasks. In systems involving hardware acceleration this can have a larger impact than when using traditional sequential computation, due to the overhead of switching between processing tasks. This closely resembles work done in the AI community regarding task allocation between different agents (e.g. the work done by King *et al.* [118, 119]). This commonly involves introducing a bias towards different tasks for a specific agent, or relying on emergent behaviour from decentralised goal-driven agents [14].

2.1.2. IoT System

The second application scenario we consider is a typical IoT system such as a smart home system. Most of the data sources in these systems are relatively simple – with the possible exception of smart cameras which resemble the use case in Section 2.1.1. The complexity in these systems comes from having a huge number of devices and therefore data sources, which can increase the complexity of balancing the workload efficiently between the available devices and servers. Since these systems may be expected to scale indefinitely (as is the case with smart cities where a single system could cover an entire city), managing them can be highly dynamic and complex.

Another issue common in such systems is that it often relies on interactions with the user, which creates real-time requirements. For example, when he uses voice commands to control his heating, or his movement is being predicted through a set of sensors in order to turn on his lights, it is very noticeable when there is latency while the request

is being processed. This creates at least soft deadlines, where tasks should be processed within a reasonable time frame to be most useful.

As an example of an application with considerable computational requirements in such a system, let's consider the collection and processing of audio as described in Figure 2.5. A number of different devices in a smart home may be capable of recording sound (1) – anything that offers voice control or activity tracking through sound – which can create a large number of similar data sources that need to be processed. Each collected sample may be useable by multiple applications, creating a wider task graph where different algorithms need to be applied to it.

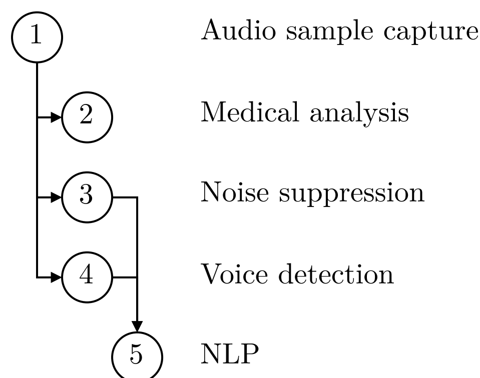


Figure 2.5.: Task graph for IoT use case

For example, current state-of-the-art technology based on different neural networks could be used to do medical analysis (2) such as depression detection [253] or activity recognition [128]. Additionally, more conventional Digital Signal Processing (DSP) techniques can be used for noise suppression (3) or speech detection (4). These may again be further processed by other algorithms such as NLP (5), which might be dependent on the results of another step – e.g. only being performed when the voice detection indicates the presence of a voice. Reducing the executions of more complex tasks not only reduce overall processing required, but also the communication overhead from offloading or distributing them. Optimisations of these task graphs can be very beneficial when using reconfigurable embedded devices [52], but is considered outside the scope of this work.

This scenario is more dynamic than the previous, as devices can freely join and leave the system. Workloads can also vary based on the changing requirements of the user, causing unpredictable and varied workloads. The objective of heterogeneous embedded devices is then to provide adaptivity by deploying hardware accelerators only when required. Optimising these systems is particularly challenging when dealing with larger systems, since centralised systems cannot solve this typically NP-hard problem [40, 45]. Therefore, a decentralised approach that can self-adapt is crucial – even when only optimising locally instead of searching for the globally optimal solution.

2.2. System Model

In our system we have devices, hardware accelerators, tasks, and agents. Here we will create a model of our system by defining what we mean by each of these concepts, and setting out any ***assumptions*** we have made. Through this, we will create a system model that describes our system.

Each *device* in our system comprises of at least an MCU and an embedded FPGA that are locally connected and can freely communicate with each other. While the MCU offers highly efficient basic processing and application logic, the parallelism offered by the FPGA can be used to perform more computationally expensive operations. In order to communicate with other devices and services in the system, we also assume that ***each device also incorporates a local wireless transceiver*** that allows it to send and receive data.

While the MCU operates by executing a sequential software program, *hardware accelerators* specify the way the FPGA can process data. They define the connecting and configuring of many small computational building blocks inside an FPGA to allow it to instantiate practically any arbitrary computing architecture.

At the heart of our system is the concept of *tasks*. The computations and processing that an application requires are modelled as a set of tasks, each of which can (at least conceptually) be performed on the local device or offloaded to an edge or cloud server. Performing the more complex computations locally requires hardware acceleration, since we assume that ***these tasks are too complex to be performed on resource-limited MCUs***.

We associate each task that needs to be offloaded to a local FPGA with a specific hardware accelerator that solves it (also referred to as a configuration). This assumes that ***only one accelerator can be executed on the local FPGA at a time***, since we use embedded FPGAs that have limited processing resources. However, extending our system to include larger FPGAs would not be infeasible. It should be noted that smaller FPGAs can be more complex to manage, as they typically do not include the self-managing componentry of larger components.

This highlights the importance of developing optimised accelerators in order to utilise the available processing power as efficiently as possible. Although we assume that ***a generic accelerator design may be available***, its adaptation to our system architecture is considered within the scope of this work.

We also assume for now that these ***accelerators are available to the device on local storage*** (e.g. a flash chip) at runtime. This also assumes that the device has enough storage available to store all of the configurations it will need in an application. We did perform a proof of concept for deploying new configurations to the device Over-The-Air (OTA) using a simple 802.15.4 interface [243], but consider integrating this into the larger system outside of this work's scope.

For each task to be solved, a number of decisions must be made, such as who is performing the computations required and whether to address it immediately or leave

it for later. Responsible for the decisions on how to handle tasks is an *agent* that may be either device-local or shared within the system.

Furthermore, we assume that all embedded ***devices are energy and/or resource-limited***. We therefore consider each device to be semi-tethered to a stable power source (i.e. dependent on a battery but capable of being recharged when required). This means they need to use energy sparingly — emphasising computational efficiency.

In search of energy efficiency, we emphasise the need for resource efficiency – i.e. utilising the available computational building blocks as well as possible. Since smaller FPGAs are cheaper and have less static power consumption [51], we want to use the smallest FPGAs possible. This means minimising resource overhead introduced by our system, and optimising our accelerator design to need fewer resources.

Figure 2.6 shows a setup with three cooperating devices sharing tasks. Device 1 is sending a task A to device 2, so its FPGA can remain sleeping while device 2 already has the correct configuration loaded. At the same time, device 2 is sensing data and creating a different task (task B) which it may offload later to device 3.

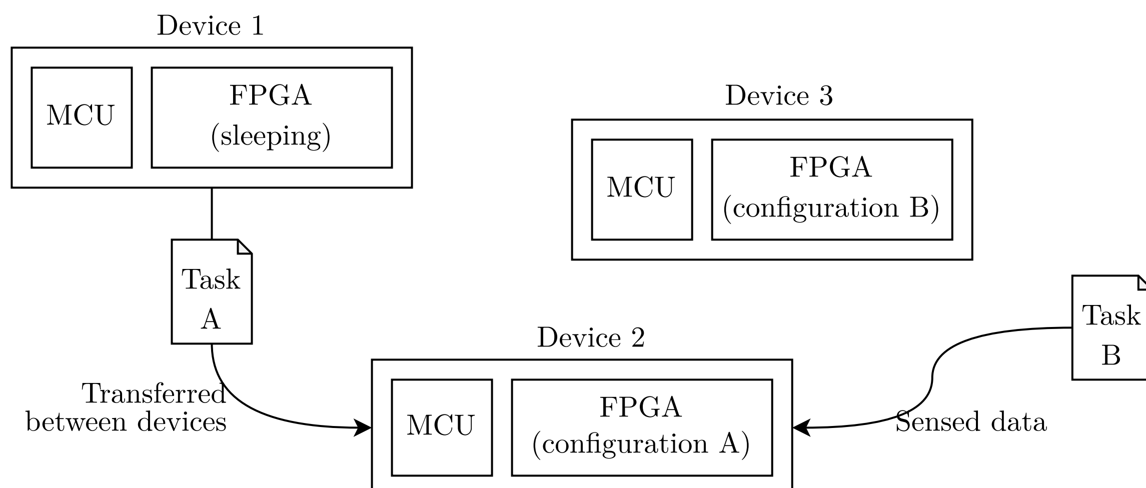


Figure 2.6.: System model showing three FPGA-augmented devices, with Task A being moved from Device 1 to Device 2, and Task B being created by Device 2 from sensed data.

The role of the set of devices is then to perform these tasks within certain limitations such as available energy, real-time deadlines, or wireless bandwidth. Therefore, we rely on high-level user goals that direct the devices' behaviour. This can take the form of a set of requirements for an application (e.g. survive as long as possible on battery power, or monitor a physical phenomena as accurately as possible), which need to be balanced and optimised.

We assume that ***tasks are created randomly by all devices*** in the system, and ***any device is capable of deploying the required FPGA configuration*** to solve

it. Although this sounds somewhat homogeneous, at each timestep each of the devices can be in any of a number of different configurations. We also assume that ***the devices in the system are cooperative***. Although they may each have their own objectives, we exclude possible security concerns such as attacks from compromised or antagonistic devices. This implies that ***all devices are trustworthy and known***.

Although devices may join and leave dynamically, we assume that ***all devices know which devices are available and can be reached***. This may be system-global in the case of a home network where any device communicate with any other device directly, or device-local peer-to-peer where only a subset of neighbours can be reached. Multi-hop communication with variable latency between different devices is ignored, but could be included by considering message transmission time.

2.3. Typical Implementation

We envision a system where every device deployed in a system is augmented with hardware accelerators on local reconfigurable hardware such as an FPGA [211]. This allows them to address considerably more demanding applications through local AI, as well as also embrace the sentiment of self-x and autonomic computing systems by creating dynamically adapting systems of devices where each device can adapt to perform different roles.

Note that this does not assume a homogeneous system where each device has the same hardware make-up, or even behaves in the same way. King *et al.* showed the advantage of having heterogeneous systems [119] where devices can adapt to locally changing requirements. Our goal is to have each device be capable of managing its own role in a large, complex system in order to scale adequately for large-scale systems. If all the devices in a system depend on some centralised control paradigm (e.g. centralised observer-controller models [28]) it would be unreasonable to expect the system to scale to the size of a smart city.

One of the major focusses of this work is the development cycle, as this currently poses a major obstacle to adoption of FPGAs in embedded systems. To make our vision of ubiquitous augmented devices a reality, the effort required for development must be reduced drastically from the manual hardware/software co-design process it is today. This means firstly that additional support should be provided to developers, and secondly that services should be provided that can orchestrate such a deployment (this includes our proposed microservices-based solution that is largely outside the scope of this work [33]). Simply put, our goal is that developers should be able to utilise hardware acceleration on these devices as simply as utilising a software library.

The system of FPGA-augmented devices relies on a very complex development paradigm as shown in Figure 2.7. On the local device, it depends on a hardware accelerator developer (1) to design the hardware accelerator on the FPGA. This would commonly either be in a Hardware Description Language (HDL) like Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog, or using system designer

tools such as Embedded Development Kit (EDK) or High-Level Synthesis (HLS). Regardless, the result would be a synthesisable architecture design that can be instantiated in the reconfigurable logic of an FPGA.

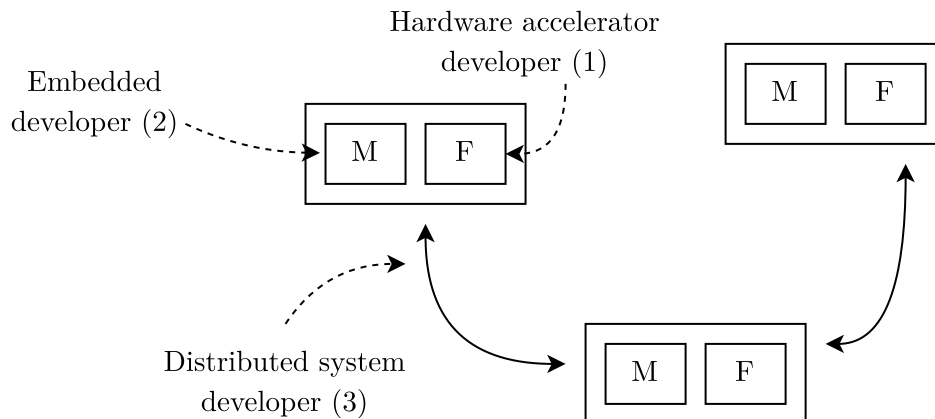


Figure 2.7.: System design process overview showing each involved developer. Hardware accelerator developer (1) creating designs for the (F)PGA, Embedded developer (2) focussed on (M)CU programming, and the Distributed System developer (3) creating connections between devices.

Next, the embedded software developer (2) creates the application software that runs on the device. This commonly includes basic sensor and actuator integration, and data management such as communication with a central server or peers. Likely this would take the form of a set of libraries, as well as either bare metal application logic or incorporation into the local embedded Operating System (OS) (either an embedded version of Linux or an embedded Real-Time Operating System (RTOS)). This creates a functional device that can take advantage of the provided local hardware acceleration.

Lastly, the distributed system developer (3) is responsible for integrating a number of these augmented embedded devices into a system. This involves the distribution of an application through offloading or a similar technique. Device AI or other self-x techniques such as self-organisation or autonomic self-awareness are also introduced here (which will be further discussed in Section 3.5.1), leading to a set of devices that can collaborate towards a larger goal or application than any of them could achieve individually.

Our goal with this work is to provide support for all three these developers. This involves a suitable hardware platform for this device, providing support for local development of application logic on it, and demonstrating how device intelligence can create collaboration. To solidify these goals, we need to define our system requirements that will be used to evaluate our result and to address our research questions.

2.4. System Requirements

We detail here the functional (F) and non-functional (NF) requirements for the overall system. These requirements are written with the needs of each of the developers discussed in Section 2.3 in mind, and serve to highlight the main issues that need to be addressed if we hope to answer our hypothesis. They direct our design decisions throughout this work.

SREQ_{I}: (NF) Adequate Local Intelligence to Support Various Applications

The embedded device needs to offer enough local intelligence to support the processing requirements of complex applications that rely on AI. As stated in Chapter 1, we plan to bring down intelligence from the cloud to the device. This puts additional computational strain on the local device, since it now needs to do more than in a traditional IoT system where it only collects data and forwards it to some central cloud service for processing.

How much local intelligence is required will be application-dependent, impacting this requirement. Even a single application's requirements may change over time, emphasising that we need the local device to be adaptive. In general, we require the device to be capable of adequate local intelligence that it can support demanding applications that rely on AI techniques such as CV or ML. A local hardware accelerator should be capable of at least comparable processing speed as the alternative – e.g. offloading the task and processing it in the cloud.

SREQ_{II}: (F) Automated Decentralised Cooperation Optimisation

A single device is often not enough to sufficiency solve more complex applications. Therefore, distributed embedded systems solve problems that are too large or impractical by splitting the problem into smaller chunks. Doing this is non-trivial, especially when the application requirements or set of available devices can change. If the devices are able to optimise this themselves, they can dynamically adapt to these changes without requiring additional effort from the developer.

We require that our devices should automatically form teams and cooperate to solve a problem, and optimise both collective and individual behaviour. This implies additional intelligence in the form of a range of *self-x* concepts (see Section 3.5.1). These include some awareness of their own performance and environment, and the ability to adapt to unforeseen changes through autonomous and continuous self-optimisation. This could include external disturbances (e.g. changes in weather or network availability) and changes in the team (e.g. devices joining and leaving unexpectedly).

SREQ_{III}: (NF) High Energy Efficiency

Creating a system that maximises its energy efficiency is critical, as IoT and distributed embedded applications often rely on battery power or resource constrained devices. Efficiency here refers both to low power usage through limiting the amount of energy used in a specific time frame, and efficient utilisation of available processing for maximised performance. This encompasses the two sides of the energy efficiency coin, where performance and power usage must be carefully balanced to achieve our primary criteria for high energy efficiency: low energy usage per task.

Apart from a conceptual design goal for "efficient systems", our objective is also linked to our desire for realistic experiments [33]. We need a real-world experimentation platform to add validity to the experiments that can be performed using our system. Instead of being limited to simulations and modelling, this would allow a user to evaluate both their created applications and system optimisation techniques on real hardware before deploying their solution — providing them with a more realistic evaluation grounded in real world data.

To accomplish both of these efficiency targets, our system needs to be applicable for deeply embedded devices that have very restricted resources. Due to targeting low cost and power usage, this limitation applies to available total energy, memory, and computational resources such as MCU frequency or FPGA configurable logic blocks.

SREQ_{IV}: (F) Convenient and Efficient Local Hardware Accelerators

Applications may require multiple different FPGA configurations: either for solving different tasks or as distinct processing steps within a single task [51]. We identify two objectives to achieve this: convenience and efficiency of switching between different hardware accelerators *at runtime*. The switching efficiency is directly linked to the switching latency due to the energy overhead involved in changing the active configuration on the FPGA.

The second part of the requirement is that switching and managing accelerators at runtime should be simple, pertaining to the application development and the software runtime we need to provide. Our focus here is on simplifying the application design process to incorporate multiple accelerators into an application. Managing the FPGA includes turning it on and off, exchanging data with it, and keeping track of locally stored accelerators. Our goal is to have hardware accelerators incorporated in a similarly simple fashion to software libraries in the Arduino development environment [8], where one only has to include the relevant header file and call the relevant function.

These requirements highlight a number of distinct elements in our system. Firstly,

the two levels of AI – local application intelligence and shared cooperative intelligence – are both required for solving the complexities of smart IoT devices. These devices also need to be energy efficient, leading to restrictions on the available resources. Lastly, creating applications that may involve multiple accelerators requires design support for incorporating them into the application logic, and runtime support for managing the FPGA.

The next step is to clarify some fundamental concepts that we need to satisfy these requirements in Chapter 3. This will clarify some terms already used in this work, as well as detailing the additional ones we will need for the design of our system. Once this is complete, the first of the three phases will be presented in Chapter 4: the design of the appropriate hardware and software runtime for our work.

In this first phase an additional set of requirements will be created in Section 4.2, specifically focussing on our hardware platform/software runtime. Later, another set of requirements is detailed in Section 6.3 to ensure that the learning agent that controls the higher level behaviour. These additional requirements serve alongside the system requirements to clarify the role of these two components of the system – the platform and the learning agent.

Chapter 3.

Fundamentals

Our system as presented in Chapter 2 lies on the intersection between a number of different fields. Creating autonomic FPGA-augmented smart IoT devices requires a good understanding of heterogeneous embedded hardware, distributed embedded software, and general reconfigurable hardware accelerators. Additionally, giving them local intelligence for optimising their own behaviour relies on both the fields of learning intelligent devices and machine learning. Each of these fields are built on a number of fundamental concepts that underpin the scientific work done in it. Therefore, we discuss these concepts here to clarify any unfamiliar or vaguely defined terms. A study on specific implementations of these techniques follows in Chapter 8, where we discuss approaches followed by related works in the literature.

For brevity, the areas being discussed have been combined into five distinct topics. We start by discussing how embedded systems can be augmented with hardware acceleration in Section 3.1 (covering both hardware and software), and how they can be optimised in Section 3.2. This is followed in Section 3.3 with a discussion on different application models for creating heterogeneous applications.

After this, Section 3.4 provides some examples of connected computing paradigms that offer different ways for multiple nodes or devices to share an application. Lastly, a higher abstraction is discussed in Section 3.5 regarding intelligent devices that learn – providing some techniques that can be used to increase the intelligence of a device's decision-making.

3.1. Accelerating Computation in Embedded Systems

Increasingly complex applications in the IoT and mobile embedded devices rely on new methods of computation. Instead of the basic computation abilities of small microcontrollers with low power consumption, techniques from other fields such as AI and machine learning are becoming more pertinent in embedded systems. This brings with it tremendous possibilities for local intelligence, but also raises complexity by imposing new requirements and restrictions. While hardware acceleration has dominated data centre and commercial computation over the last few years [278], it has only recently become feasible within the power and size constraints of the IoT [201].

We discuss here some of the options for incorporating accelerated computation into

embedded systems (both hardware and software acceleration are considered). Some of these are already fairly mainstream, while others have only seen very limited adoption.

3.1.1. Multicore and Multithreaded Systems

Likely the most ubiquitous approach to increasing processing performance is using multicore systems, where a single CPU or MCU includes multiple physical computing cores. Each of these cores is capable of performing somewhat independent instructions, while commonly still sharing some resources (such as local cache memory). Multicore processors generally take one of two forms, where the cores are either used in unison or entirely separately. Especially in the field of DSPs, multiple cores can collaborate to perform Single Instruction Multiple Data (SIMD) instructions that can process multiple inputs at the same time (e.g. Super Harvard Architecture Single-Chip Computer (SHARC) DSPs used primarily for audio processing [102, 235]). This can further be extended with special instructions that can directly process a vector (e.g. the Advanced Vector Extensions (AVX) processor instructions found in most mainstream desktop CPUs [131]).

Much work is being done in the field of multi-core processing[1], focussing on designing both the hardware and software designs for creating SoC solutions with multiple processing cores. Therefore, creating applications that take advantage of the linear scaling that comes from using multiple cores has become considerably simpler over the last few years, relying on sophisticated compilers and abstractions rather than custom assembly programming of multi-core solutions.

Related to this is the use of multi-threading, where the CPU is capable of processing a number of different threads of execution at the same time [221]. In terms of processing acceleration this means that in a multicore system the CPU can perform numerous independent tasks at the same time. However, in essence multi-threading simply relates to being able to switch context between various “threads” (or “tasks” as they are commonly referred to in RTOSs such as FreeRTOS[2]).

The important task when developing multithreaded applications is to isolate and identify each of the threads, enabling them to perform arbitrary tasks simultaneously – without requiring excessive synchronisation points or passing data between them. This multithreading functionality is usually provided by the OS, as each thread of execution needs to be managed and controlled. Our aim for this work is to be OS-agnostic, meaning that our solution should be compatible with an embedded OS such as embedded Linux or an RTOS – without being dependent on it. Therefore, we do not include multithreading into our design, but rather would consider our delegated hardware acceleration as a single task or thread that could be managed by the OS.

[1] `https://www.mcsoc-forum.org` (last visited: 2021-12-06)
[2] `https://www.freertos.org` (last visited: 2022-03-13)

3.1.2. Field Programmable Gate Arrays (FPGAs)

On the scale of application specificity versus general purpose processing, the range is commonly considered to be from Application Specific Integrated Circuits (ASICs) that are designed to solve one specific problem as optimally as possible on the one extreme, to general purpose CPUs that aim to solve every computational problem through a variety of instructions [215]. On this scale, an FPGA sits much closer to the ASIC, but provides more general applicability through reconfiguration: instead of being manufactured with one permanent architecture, it can change its architecture whenever it is required. Practically, one can consider FPGAs to be ASICs that can be reprogrammed whenever required by redefining the internal connections and the configuration of small building blocks.

These building blocks are referred to as Configurable Logic Blocks (CLBs) that can be in one of a few different modes [201]. For example, the CLBs in the modern 7 series family from Xilinx [265] can be used either as a Lookup Table (LUT), distributed Random Accessed Memory (RAM) or a shift register. These blocks can then be connected at will using the programmable interconnect, which efficiently transmits signals at a circuit level from one block to another. Together, this provides users the ability to create nearly arbitrary logic circuits that can perform computational tasks by instantiating an architecture for performing all necessary 'instructions' (i.e. operations).

The CLBs are not always optimal for instantiating larger amounts of resources — which is particularly true for volatile memory. Although the CLBs can each instantiate a roughly 64 bit register, this may not be adequate for larger applications, which is why a number of additional dedicated resources are provided within the reconfigurable FPGA logic. Block RAM (BRAM) is an important example of this, and provides very fast memory blocks in between the other configurable blocks. Users can either specifically choose to instantiate one of these blocks (commonly a few kilobyte in size), or the toolchain can identify larger sections of memory that can be optimised to BRAM.

Similarly, the mathematical abilities of the system are expanded by introducing DSP elements that can perform specific mathematical operations (e.g. Multiply-ACcumulate (MAC)). These not only improve the complexity of the architecture that can be instantiated by freeing up the more generic CLB resources, but also increases performance as these DSP elements commonly operate at a higher clock speed than the CLBs themselves.

Designing an architecture in this way is traditionally done using an HDL such as VHDL, effectively manually defining each interaction and connection. This has been shown to be very time-consuming and lead to large development overheads [55], making it less attractive to developers than mainstream alternatives such as GPUs [185, 188] discussed below. Similarly to conventional programming languages such as C, basic operations such as multiplication can be used using

```
a <= b * c;
```

where each variable has a defined type (e.g. a signed number). A fundamental difference from conventional programming languages is that here a is defined to *always* be equal to the product of b and c, instead of the instruction being computed at a specific point in time. This means that if any of these input variables change, the value of a would be updated *immediately* and *implicitly*. Although this leads to very high computational performance, a multiplier circuit would need to be dedicated specifically to this equation, and not used for any other operation in the lifetime of the architecture.

A number of more modern approaches to developing for FPGAs have been introduced [185]. These include products from mainstream brands such as Xilinx, Altera, and Matlab. These commonly offer developers the ability to create systems in easier to use languages, and then later compile them into usable HDL. These systems see a regular rise in popularity, but the plethora of options available can make it daunting to approach for beginners, as most of them are designed with a specific use case in mind (e.g. the signal processing code generator "HDL Coder" from Matlab [153]).

3.1.3. Graphics Processing Units (GPUs)

A more mainstream hardware acceleration option is the GPU, which offers massive parallelisation through a defined SIMD architecture [185]. This allows it to perform the same operation to a large vector or matrix of input, making the GPU particularly proficient at processing uniform data streams such as images or video [149].

In the embedded space, GPUs have been used primarily in mobile phones that offer a similar system architecture to conventional desktop computers by combining a CPU and a GPU into a heterogeneous system. The GPU is primarily used to accelerate the rendering of imagery and the interfacing with the display, and has been used for hardware acceleration mainly in video games and similar applications.

Due to limitations in the support for GPU accelerator languages such as CUDA [172], custom hardware acceleration has seen very limited use in the mobile device community. Special hardware platforms such as NVidia Jetson [173] have been introduced to promote using traditional GPU acceleration in the embedded space, but these are limited by having considerably larger power usage than is practical for battery-operated devices such as wearables (since the Jetson line is targeted primarily at the automotive industry).

3.1.4. Heterogeneous Computing

A popular approach to developing accelerated computational embedded systems is to combine different processing units into one heterogeneous system – in contrast to homogeneous systems where all of the processing units are the same. For example, a popular commercial implementation of this is the ARM big.little [9] design, which is a multi-core system that consists of different types of cores – some smaller and low powered for simpler tasks and some more powerful for more demanding computational loads. It has been used very successfully in mobile systems such as ARM-based phones or laptops (e.g.

new M1 laptops from Apple license processor cores from ARM[3]).

Another version of heterogeneous systems can be found in approaches that incorporate a variety of sequential and hardware accelerator cores [49], such as combining a conventional CPU and GPU. These have the benefit that each task can be individually assigned to the most appropriate processor, leading to improved efficiency and greater performance overall.

In the embedded sphere this can be either a number of discrete components, or one combinational component (e.g. an SoC). Apart from the CPU-GPU combinations commonly found in mobile phones, the most popular Commercial off the Shelf (COTS) embedded device is likely the Xilinx Zynq-7000 family [268] that combines a single or dual-core ARM embedded processor with Programmable Logic (PL) in the form of an FPGA (e.g. [90, 130, 155]). This FPGA is on par with either a larger Artix-7 or Kintex-7, providing considerably more resources than the smaller embedded ranges such as the basic Spartan FPGAs. It offers highly convenient SoC development, as the ARM cores and the PL can easily interact through the use of a provided Advanced eXtensible Interface (AXI) bus. This creates a relatively beginner-friendly entry point into designing heterogeneous systems that utilise FPGA acceleration.

3.2. Optimising Hardware Acceleration Performance

Efficiently utilising hardware acceleration requires different optimisations than used in traditional sequential computing. Although multithreading and multicore design can increase the number of parallel tasks, that mostly scales performance linearly (at best) with the number of available cores and the frequency. This makes optimising their performance fairly straightforward – ignoring some losses due to switching threads and frequency ramping [152].

Therefore, we present here some general optimisation techniques for improving the performance of hardware acceleration. These optimisations are particularly crucial in embedded applications, as energy consumption must be carefully managed – requiring the maximising of performance and minimising of power usage.

3.2.1. Pipelining

Hardware acceleration commonly relies on complex pipelines for ensuring that the available resources are fully utilised. As shown in Figure 3.1, this ensures that all available hardware can work at the same time by processing various steps from subsequent tasks. In the ideal case, this allows the architecture to create one output as every input is provided to the system.

Utilising pipelining requires both the design of an optimised pipeline (where each stage requires roughly the same amount of time to process to minimise downtime) and

[3] `https://www.tomshardware.com/uk/news/Apple-M1-Chip-Everything-We-Know` (last visited: 2022-04-15)

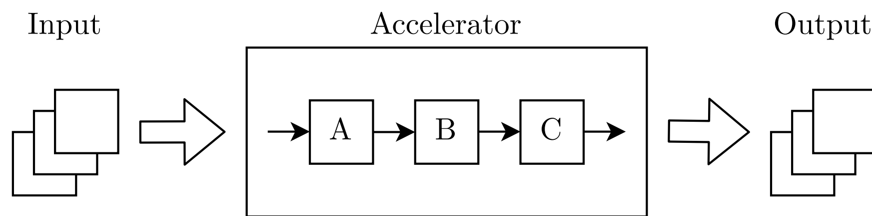


Figure 3.1.: Pipelining in hardware accelerators, showing how inputs are efficiently pushed through the accelerator to create a sequence of outputs

independent, predictable stages that do not require communication or have dependences between them. Although pipelining is not relevant to all applications, it can drastically improve the performance of multi-stage problems (e.g. CV techniques that require a number of complex sequential processing steps).

3.2.2. Parallelism

Increasing parallelism in hardware accelerators requires the processing of one element entirely independently of another. For example, Figure 3.2 shows a setup where multiple identical accelerators are deployed to a device, and each processes one set of inputs at a time. This can be beneficial when the resource requirements of the computations fall short of the available resources, and allows efficiency to be balanced with performance.

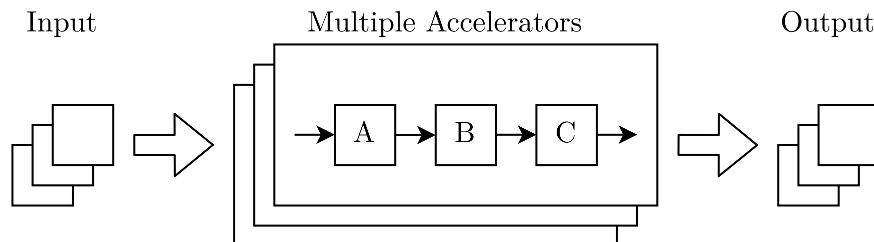


Figure 3.2.: Parallelism in hardware accelerators, creating multiple outputs at the same time through multiple deployed accelerators

While parallelism is only beneficial with adequate resources, it may be applied to a segment of a computational problem. For example, an audio application may search multiple incoming streams for keywords simultaneously using simple preprocessing accelerators, before further investigating any identified samples using a single (more complex) processing accelerator.

3.2.3. Batching

Arguably the simplest way to improve performance with using a hardware accelerator as coprocessor (where it gets assigned specific tasks by a main processor) involves batching

workloads. As shown in Figure 3.3, the inputs that need to be processed are collected and bundled together before being sent to the coprocessor in one large chunk. While this does not directly affect their processing on the accelerator, it can reduce the offloading overhead from exchanging multiple small bundles of data.

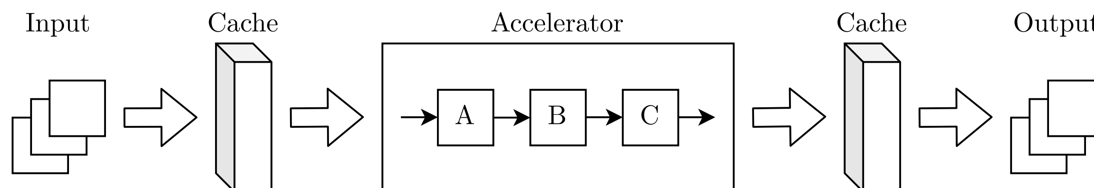


Figure 3.3.: Batching in hardware accelerators, optimising the overall performance by minimising communication overhead.

In some cases, batching can have further beneficial effects when the outputs of an entire batch is combined during processing. For example, the stability of training neural networks can be improved through batching [84]. In contrast to online methods, these systems combine the effects from the entire batch into a single update. Not only does this improve performance on parallelised hardware such as GPUs as it takes advantage of SIMD instructions like array multiplication, but it also creates a more stable system as the noise from individual updates from each input is effectively suppressed.

The reconfiguration required for FPGAs to perform a specific task can be seen as overhead. Whenever a task needs to be performed, the FPGA needs to spend a relatively large amount of time and energy reconfiguring its logic to load the required configuration [244]. This provides a good example of the benefits of batching, as each batch only requires a single reconfiguration.

We investigated this as part of our demonstration at the 16th International Conference on Autonomous Computing (ICAC) [211], where we showed the effect of changing the batch size when offloading to a local embedded FPGA. Figure 3.4 shows the total latency for performing 100k iterations of an application that requires multiple accelerators. We can see from the figure that the latency introduced from reconfiguring is reduced drastically as larger batches are used. By reducing the total number of times the FPGA needs to load a new hardware accelerator, the total latency from performing all 100k iterations is reduced. Note that the inflection point towards the 10^4 mark is when the FPGA is spending a larger percentage of time computing, even though the total duration is still decreasing.

These three techniques can be fairly trivially combined, as their implementation is either within the hardware accelerators or how they are deployed on the hardware. For example, one could have multiple parallel instantiations of an accelerator that each get fed a full batch at a time. We have seen that utilising optimisations such as these have a large impact on the efficiency and performance of heterogeneous embedded systems. Therefore, we prioritise local batches when creating the local AI that optimises

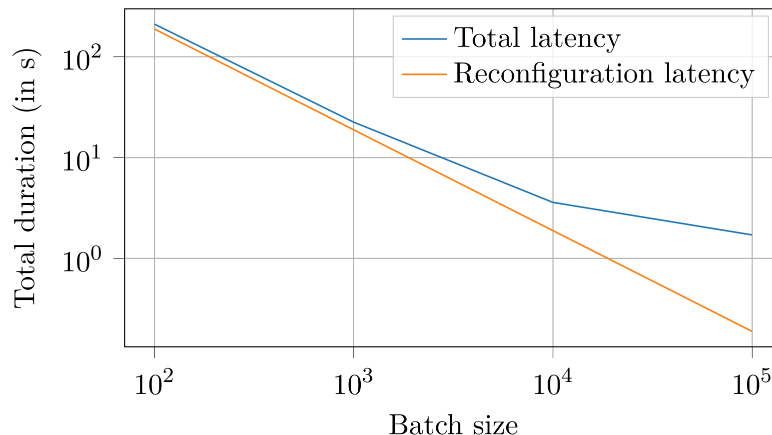


Figure 3.4.: Latency of offloading different sized batches to a local embedded FPGA. Based on our experiment of performing 100k iterations with different sized batches.

device behaviour, as we showed that efficiently balancing FPGA reconfigurations with processing is paramount.

3.3. Heterogeneous Application Models

Creating software for heterogeneous systems can be complicated, since it involves sophisticated synchronisation and information passing between the various components. Therefore, a number of different abstractions have been created for simplifying these interactions. Some of these are purely software-based solutions (such as OS threads and middleware), while others involve conceptually different system design (e.g. tile-based or SoC solutions). We describe here some of the more popular approaches (especially in the embedded field), with specific examples of projects in literature being provided in Chapter 8.

3.3.1. OS Threads

Abstracting offloaded computations as OS threads (so-called *hardware threads*) has the benefit of being familiar to many developers: any programmer that has created either multithreaded desktop or embedded applications (or even an RTOS) will be familiar with the interaction scheme of threads and the accompanying mutexes and semaphores. These are essentially thread-safe locks that provide access to shared memory or interfaces – without creating interference between the different threads. For example, in a heterogeneous system where the processing cores share external memory or a communication interface, they need to manage who is interacting with it at any given point in time.

The second benefit of threading-based application models is that it includes context, meaning that there already exists the mechanism for each task or thread to have local variables and memory. This is important when different parts of the application are running simultaneously on different processors, isolating each thread within its own context.

Lastly, threading is already supported in the case where an OS is present. When working in embedded Linux, for example, the kernel can natively manage the multitasking nature of having both hardware and software threads. This means that once an accelerator has been abstracted to a hardware thread, the system can manage the rest of the execution, synchronisation, and management.

3.3.2. Modular Tile

Although not quite as popular lately, modular tile-based application models saw great popularity in the early 2000s [148]. The basic premise is that an application is broken up into smaller modular chunks that are fully interchangeable. This then creates a very scalable solution where an application can deploy as many of these different tiles as required. For example, a number of vector-processing units may be deployed that can either process in parallel or series. By passing data to each other along either a shared bus or a peer-to-peer system as shown in Figure 3.5, they can create an interconnected collection of processing cores.

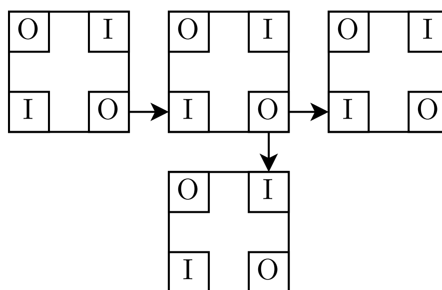


Figure 3.5.: Tile-based modular components passing data to each other using dedicated I/O blocks

One downside of this type of system is that it requires a custom programming model, since each individual core has its own instruction set. Additionally, controlling the overall system of tiles is non-trivial since commonly a message-based communication model is needed. This leads to each system requiring its own compilation system and often a unique programming language, creating a steep learning curve for new developers.

3.3.3. System on Chip

Although an SoC approach is mostly a hardware solution, it leads to a unique application model with very tight coupling. Essentially it involves designing a custom silicon chip

that incorporates all or some of the required components required – instead of installing each separately on a Printed Circuit Board (PCB) and connecting them with copper traces. Since the job of connecting different processing cores is substantially easier electronically when they exist on the same silicon, these solutions generally offer better interaction between cores than multi-chip alternatives. Additionally, this connectivity is generally faster and have lower power consumption.

While this is commonly a very good solution for industry projects aiming for high volume long term manufacturing, it is not as beneficial for experimental research. Designing a new SoC has a very long development time and cost, making it infeasible for experimenting with various different combinations of processors and peripherals.

3.3.4. Middleware

The concept of a middleware has been extensively used in distributed computing and networking. It generally creates an abstraction layer between a distributed application and the OS, or even instead of an OS. It primarily aims to simplify application development by connecting the necessary components within the system.

This allows it to present a higher-level abstraction for the programmer to use, thereby providing a generic solution to common issues such as data synchronisation and Remote Procedure Calls (RPCs). This highlights what we consider the primary goals of a middleware, namely to provide easier application development and to avoid duplication of development work. The task of the developer then becomes to make their own designs compatible with the interface of the middleware, at which point they can be combined into an application using a convenient provided Application Programming Interface (API) by the middleware.

3.4. Connected Computing Paradigms

Connecting various computing devices together requires a clear definition for how computations are divided amongst them. Numerous such paradigms are presented here for splitting the work between a system's embedded devices and larger (more powerful) elements such as servers. This also involves the important decision of whether to move a specific piece of computation to another device. Therefore, we specifically discuss here the concepts of offloading and placement.

3.4.1. Offloading

Schaefer [209] refers to offloading as focussing on the extraction and remote computation of a computationally intensive part of an application. The result of the computation is then sent back to the original application once ready. Flores *et al.* introduced a popular structure to analysing offloading in the context of mobile cloud computing that considers the four basic questions: what, when, where, and how [76]. These questions help to

understand not only the technicalities of the server-client interaction, but also the goals involved.

What? The question of what to offload generally comes down to your objective (e.g. energy efficiency, real-time deadlines, throughput). The general concept is to offload any section of the application when that will improve the system's performance according to your chosen objective. How these sections are isolated from the rest of the application comes down to the application model (as discussed in Section 3.3), which describes how an application can be broken down into separate parts (e.g. tasks, components).

The component responsible for answering this question is called the *code profiler* [76]. It may follow a manual directive such as annotations in source code [58, 122] or a more dynamic approach that uses static code analysis or history traces [48, 145].

When? This question relates to deciding whether or not an application section defined by the code profiler gets offloaded or not. This offloading decision is made by the *decision engine* [76], and can be made either statically or dynamically [48, 58, 125, 126]. Multiple factors may influence the offloading decision, conceptually related to *self-aware computing* (see Section 3.5.1) when devices are expected to self-govern.

Where? This issue can again be solved in either a static or dynamic fashion. One may consider this an extension of the When? question, where the offloading decision must be optimised in the new dimension of having different destinations available. Each option must be evaluated in order to choose any of the available resource providers (or the local case when none of them are viable).

This decision is generally based on the current state of the different agents in the system, and therefore the full system state must be monitored by the so-called *system profiler*. This introduces an additional level of complexity, where the accuracy of the offloading decision must be balanced with the availability of the system state. This is even more relevant in the case where state is very volatile, since it would be impossible for each agent to always have a full overview of the system.

Although most approaches in the Mobile Edge Computing (MEC) and Mobile Cloud Computing (MCC) communities address the binary offloading question of whether or not to offload to a single server, a few projects have approached the problem where multiple options exist. One such project by Xu *et al.* [271] addresses a simpler problem of selecting a server for processing a data stream, where the decisions remain in action for a period of time. One of a number of available server nodes are chosen to process the incoming data, based largely on their available resources. Similarly, Ma *et al.* used game theory to address the option of different access points to offload to [144].

A multi-layered approach is followed by Chen *et al.* so the device can choose between offloading to the access point directly, or through it to a cloud service [40]. Here, it appears that the mobile client only performs a similar binary offloading decision, after

which the request may be forwarded to the cloud to alleviate processing limitations on the computing access point.

How? The offloading mechanism defines how a part of an application can be executed remotely in an offloading scenario. It involves recreating the state required to perform the task, as well as the description of the task itself.

There are different ways of performing this, ranging from a direct RPC to recreating an entire virtual machine. Another approach is to use message passing (commonly in bytecode format) which describes all the necessary information. The Tasklet system is an example of this, where each offloading task includes both the necessary instructions for execution, as well as the required data [209, 210].

3.4.2. Placement

In contrast to offloading, Schaefer defines placement as a system abstraction above offloading that considers the decision of where each application segment is offloaded [209]. However, a number of other interpretations have been used in other industries (e.g. traditional distributed computing and edge computing) – commonly without a clear definition being provided. Commonly they focus more on dividing an application into various parts (e.g. tasks or components) that need to be placed on a computing provider.

We would say that placement is more focussed on the application or system, while offloading focuses on the fundamental question of "should I compute this myself or not". Importantly, for offloading the decision is generally made by the holder of the task rather than some centralised service – making it more appropriate for decentralised and distributed systems.

3.4.3. Distributed Computing

Apparently there is discontent about the definition of distributed computing, but one simple version is "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system" [230]. It generally considers the different parts of the system as either being *resource providers* or *resource consumers* [38], which highlights the emphasis of resources as a modelling scheme.

3.4.4. Edge Computing

The original intent of edge computing was to provide computing resources geographically nearer to the consumer to reduce latency [80]. It focusses primarily on bringing these resources to the edge of the local network, effectively bringing the cloud closer rather than elevating the local devices. Therefore, edge devices can vary from humble Single-Board Computers (SBCs) such as the Raspberry Pi [184] to the NVIDIA EGX Platform[4]

[4] `https://www.nvidia.com/en-gb/data-center/products/egx-converged-accelerator/` (last visited: 2021-12-13)

which offers High-Performance Computing (HPC) level performance.

Edge computing solutions usually manifest as an edge computing platform that encompasses the local embedded devices, the edge services they connect to, and the mechanism required for offloading data/computation to these services (e.g. Waggle[5]).

3.4.5. Mobile Edge Computing

Introduced by IBM and Nokia for mobile network base stations, a standard was eventually created by the MEC group from the European Telecommunications Standard Institute (ETSI). Essentially the basic idea was that it should act as edge computing for mobile networks [202]. ETSI stipulates some characteristics for MEC [186], basically boiling down to a special case for edge computing that should be independent of the rest of the network, and incorporates some contextual information such as locations.

3.4.6. Fog Computing

Originally coined by Cisco [22] as an extension to cloud computing, fog computing was specifically targeted at assisting the emergence of the IoT [23]. It was defined to be "a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centres, typically, but not exclusively located at the edge of network" [22]. It evolved to become more independent of the cloud [168, 274, 275], however, which eventually led to a alternate model where it acts as a layer in between the clients and the cloud [141]. This was due to the fog devices themselves becoming more heterogeneous [202], based on the complexity associated with this independence.

3.4.7. Peer-to-peer and Grid Computing

Other paradigms exist in the world of distributed computing for collaborative applications – e.g. Peer-to-Peer (P2P) [108, 251] and Grid Computing [146, 214]. These focus on managing heterogeneous resource providers and consumers across different nodes (commonly desktop or mobile devices). In the case of grid computing, this is commonly done through virtualisation [257]. Although most applications utilising these (relatively traditional) paradigms require high-level computing resources such as desktops or servers, work has been done in smaller devices – e.g. the Tasklet system by Schäfer *et al.* [210] that includes mobile phones and to some extent embedded devices.

3.5. Intelligent Devices

The concept of intelligence in devices is tightly coupled with rationality [206], which concerns their ability to act in a way to achieve the best (actual or expected) result.

[5] `https://wa8.gl` (last visited: 2021-12-13)

Simply put, it requires them to "do the right thing". Extending this to the concept of an intelligent device suggests that the device should be able to *learn* how to act correctly. This further requires some ability to evaluate its own performance, and to adjust its behavioural strategies to maximise this performance.

3.5.1. Self-x Computing

Importance is placed understandably on the definition of *self* [212] – essentially what is defined as being *inside* or *outside* the system. Depending on whether certain actors or the user are included in this definition, a wildly varying set of systems may be defined as self-x. Commonly this rests on the assumption that a self-x system is at least capable of fulfilling some purpose entirely without external intervention, depending on which specific self-x paradigm is being referred to.

3.5.1.1. Self-Awareness

Self-awareness has been defined by Rinner *et al.* as "a system's ability to obtain and maintain knowledge about its state, behaviour, and progress" [198], which allows it to autonomously respond to changing internal and external influences. This awareness can be considered separately from the ability to act upon this information, which has been labelled as *self-expression* by Lewis *et al.* [133]. This allows us to separate the device's conceptual sensing of its situation with its ability to enact a specific change upon it.

Computational self-awareness is tightly linked to autonomous computing, which was originally presented by IBM [116]. It is based on self-governance in social and economic systems, and aims to create "computing systems that can manage themselves given high-level objectives from administrators". This highlights the need for specific objectives or utilities to be optimised, which are commonly application-dependent and therefore provided by the system designer. It has also been extended to manage a model based on its sensor inputs, and detect when this model does not match its observations [68].

3.5.1.2. Self-Organisation

Many different definitions of self-organisation has been provided in various different fields, an overview of which has been provided by Schmeck *et al.* [212]. Commonly involving statistical entropy, effectively aiming at achieving a higher level of order without external influence. It commonly handles a system's ability to manage itself in the absence of outside control [164, 212].

This leads to the simple definition that a system is considered purely self-organised if it can structurally alter itself to increase order – without external intervention. In some cases this rigid definition is impractical, as is the case in organic computing (see Section 3.5.2) where optional external direction must be supported. Related to this is *emergence*, which is defined by Mnif and Müller-Schloer [158, 165] as self-organised order.

Another definition is provided by Tomforde *et al.*, who defines a self-organising system as one that consists of autonomous entities capable of deciding how they should interact [239]. The basic objective of such a system is then to create and maintain its structure, which the authors model and measure by considering the changes in communication between the various entities in the system. This is based on the system having to communicate internally (specifically ignoring communication with centralised servers and/or external stimuli) in order to alter its own structure.

3.5.1.3. Self-Adaptivity

In contrast to this, self-adaptivity relates to a system's ability to self-regulate itself to remain in an acceptable state [212]. Even in the presence of external disturbances, it should be able to act in such a way that it continues to survive and even move into a better state (the set of which is either the acceptance or target space). Connected to this are the concepts of *robustness* (the ability to remain in an acceptable state as the system or environment change) and *flexibility* (the ability to respond adequately to changing target and acceptance spaces) [16, 212].

An example of such a system is presented by Tomforde and Goller, who attempt to quantify a system's efforts to self-adapt [236]. They also consider the rates of adaptation as this may affect user acceptance in certain applications where users interact closely with the system. Other approaches are discussed by Krupitzer *et al.* [124] in their survey on self-improvement in self-adaptive systems. They classify 19 different approaches based on criteria such as the level it operates at (e.g. application/managed resources) and whether it is reactive or proactive.

3.5.2. Organic Computing

From the self-x and autonomic computing communities arose the concept of *organic computing* [166], which frequently takes inspiration from natural and biological systems [167]. It is strongly related to the concepts of robustness, flexibility, and adaptivity discussed previously. Specifically, a system's ability to adapt autonomously and dynamically based on input from its sensors has been used to define an organic computing system [240].

At its core lies its connection to self-organisation, but it draws a crucial distinction: instead of insisting on a fully autonomous system that operates without any intervention, it allows reporting back to the user and the resulting user feedback. For example, consider an assisted automotive braking system that can autonomously adjust to current road conditions. Simultaneously, it also provides feedback to the user so they may adapt their inputs and respond appropriately. This concept is termed *controlled self-organisation* [28], highlighting the balance between autonomy and user-influence.

An important concept in organic computing is the observer/controller architecture [28, 167, 238]. It defines two distinct entities that interface with the system (SuOC). The *observer* captures the current state of the system, while the *controller* directs its behaviour. This can either be a single centralised solution that manages the entire system,

or a decentralised local solution where each part of the system has their own.

3.5.3. Autonomic Computing

Amongst self-x systems exists the concept of autonomic computing [127], which refers to the desire for a system to reliably operate entirely without external intervention. Arguably the most important concept within this is the Monitor-Analyse-Plan-Execute over a shared Knowledge (MAPE-K) feedback loop, which describes the process taken in such systems to autonomously exist. It describes how the system can reason about its current state and even judge the effects of its actions.

When originally introduced by IBM, they envisioned an elaborate collection of connected systems that each self-govern [116]. Harkening to biological systems that operate entirely without active thought or control from the host, this concept includes considerations of numerous self-x concepts (e.g. self-optimisation, self-healing etc.).

Along with organic computing, autonomic computing finds its roots in Tennenhouse's *proactive computing*, which lay much of the groundwork for systems capable of performing complex tasks without human intervention [232]. A well-designed computer system can respond faster and scale better than a human-focussed one, highlighting the importance of intelligent pervasive as well as embedded computing.

3.5.4. Machine Learning Techniques

There is no denying that ML and AI techniques have dominated recent research in numerous computer science and engineering fields. Instead of developers having to fine-tune and micromanage their systems' designs, ML allows the system *learn* its behaviour automatically. A good overview of different approaches from the literature for utilising ML for self-x and other intelligent computing systems was published by D'Angelo *et al.* [62].

An important concept with ML is *optimality*, which commonly relates to maximising value or reward [112]. In most cases, truly optimal behaviour cannot be guaranteed as it cannot be distinguished from local minima. In most cases, a best effort is made through duplication — training many models in order to achieve a best effort optimality. By instantiating each model with different initial parameters, various different results can be learnt using the same techniques.

ML learning techniques are occasionally broken up into *supervised* and *unsupervised* learning – primarily depending on whether the system is learning from a known and labelled dataset or not. A supervised learning algorithm gets provided with correct labels that it can then compare its output to. In contrast to this, unsupervised algorithms learn from unlabelled data and must therefore infer unknown structures.

The advantage of unsupervised learning is that it can learn directly in the field during deployment, instead of being trained exclusively beforehand. Although generally easier/more convenient to train offline (at design time), online systems can further improve

their performance and behaviour by learning in a real deployment. Especially in scenarios that can be highly unpredictable or difficult to model, this provides tremendous advantage over systems exclusively dependent on training on fully labelled data [17].

There also exists a hybrid solution called *semi-supervised learning* [39], which aims to combine these two approaches. For example, it might include unlabelled samples into a training set of labelled data. Alternatively, it might be known that certain samples are part of the same classification [245] – even if an appropriate label is not present.

3.5.4.1. Reinforcement Learning

Sutton and Barto suggest a third type of technique at the same layer as supervised and unsupervised learning: *reinforcement learning* [228]. Although it also learns without provided labels, it does not aim to find structure in provided training data in the way that unsupervised learning does.

Instead, the foundation of Reinforcement Learning (RL) is to provide a way for an agent to learn through trial-and-error [112] by considering the effect that a chosen action has on its *reinforcement scalar*. Sutton and Barto highlights the two most important parts of RL to be "trial-and-error search" and "delayed reward" [228]. This in turn refers to the agent exploring possible actions without guidance about which would be 'correct'. It has to try available options and attempt to gauge their validity.

The important question in RL is whether the action chosen was the opportune one, based on the *reward* r received. Sutton and Barto continues to define this reward as the numerical optimisation criteria that the agent tries to maximise. Unlike other options that only target the immediate reward received, RL is one of a class of approaches capable of maximising the long-term or delayed reward. By choosing a sequence of actions correctly, it can achieve a better performance than short-sighted alternatives.

Each action is chosen based on which state the agent is in. This can include a representation of both the device's internal (e.g. memory consumption) and external (e.g. sensor inputs) state. Along with the *reward function* $r(a, s)$ that provides feedback on chosen actions, the *value function* $v(s)$ dictates how desirable a state is. This also considers any future states and their rewards that are likely to follow, providing an outlook on which state will offer long term reward.

This leads to a *policy* π that dictates which action to take when in a specific state. It fully describes the behaviour of the agent. In the case of tabular RL approaches, this is connected to the *state-action* pairs which connects each state with the possible actions available. Each intersection of state and action then is described by its value $q_\pi(s, a)$, which is used in that policy to choose the most beneficial action for a given state.

By exploring the possible space of state-actions to find the ideal action to take for any given state, an agent can maximise its long-term reward. In learning algorithms the objective is then to strive the current policy π towards the *optimal policy* π_* that has value $q_*(s, a)$ – even when true optimality is practically never achieved and can be difficult to prove [228]. Additionally, since the true value function $q(s, a)$ is usually not

known, an estimate $Q(s, a)$ is formed.

Arguably the most interesting subset of RL techniques fall under the umbrella of *temporal difference* [228]. These techniques consider the differences in reward of different temporal sequences (over time) of input samples [233]. This is of particular interest to us since it maps easily to a device experiencing a sequence of states (inputs) over time, as these techniques can also be used to generate a sequence of control signals.

One final important distinction within RL is between on-policy and off-policy learning, which differentiate based on the policy that is being learnt. On-policy refers to techniques where the same policy used to generate actions is being trained, while off-policy train a different policy (target policy) than the one directing the agent's actions (behaviour policy). Off-policy techniques have the distinct advantage that the behaviour policy can freely explore (inevitably acting in a non-optimal way) while it is still learning what the true optimal policy is.

Sarsa As an example of on-policy control, consider Sarsa [205]: so-called due to its consideration of the current state and action (s_t and a_t), and the coming reward, state and action (r_{t+1}, s_{t+1}, and a_{t+1}). Specifically, the value update function [228] is given with

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) \tag{3.1}$$

where each Q represents the value of action a in state s at time t. This equation can then be used after each episode to update all visited state-action pairs, but cannot be directly used to choose actions during the episode as it relies on future knowledge.

One extension on basic Sarsa is *Expected Sarsa* which instead uses the likelihoods of each state-action transition to compute the *expected value*. This makes it much more computationally expensive than Sarsa, but performs considerably better in terms of rewards achieved [228].

Q-learning One approach to learn the reward maximisation off-policy is Watkins's Q-learning [254], which specifically deals with incrementally maximising the reward through accumulating a Q value. This provides certain guarantees, such as tending to an optimal accumulated reward in an infinite run [112]. It specifically addresses problems described by Markov Decision Processs (MDPs) [255] — meaning that the next state depends only on the current state and the action decision.

This leads to the formal format of the Bellman Equation [228] which computes the accumulated Q_{t+1} of

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{3.2}$$

where the decaying learning rate at time t is given by α. The discount factor γ usually balances immediate rewards against later ones. The distinct advantage of Q-learning is

in the maximisation over a, indicating that the most opportune action is used to update the state-action value, instead of the future action as in Sarsa.

Double Q-learning uses two different Q functions Q^A and Q^B that are cross-learnt from different experiences [95]. This reduces the risk of overestimation, but does not require additional training data since each decision can be based on both Q values.

Some works (e.g. [271]) use an explicit Deep Reinforcement Learning (DRL) agent in the cloud to perform the offloading decisions for all users in the system. This allows them to generalise the offloading problem to include multiple destinations for offloading, but focusses on stream offloading rather than computation. This simplifies the decision since tasks are solved very quickly upon arrival. Tremendous focus is placed on the type of communication used, which uses beamforming to create point-to-point connections. Interestingly, they optimise for the power of the Remote Radio Heads (RRHs), which process incoming streams from the users.

Actor-Critic Alternative RL approaches include Actor Critic (AC)-based options [228]. Simply put, these approaches all utilise the principle of having the *critic* estimating the value of a state-action pair (in Q Actor Critic this value approaches the Q value), and the *actor* updating the policy based on the verdict of the critic. This is a fairly universal approach in RL approaches, where the actor and critic work hand-in-hand to both utilise and continuously update the policy. An important distinction within AC techniques is that the critic can consider the state-value of the resulting state (after the action is taken), which provides an important advantage over other techniques that only consider a single state-action-reward sequence.

The two main variants of AC are Advantage Actor Critic (A2C) and Asynchronous Advantage Actor Critic (A3C) [160]. A3C implements parallel agents that learn a common and shared value function asynchronously, while A2C is synchronous as it only has a single worker.

LCS Another important technique in the field of adapting systems is Learning Classifier System (LCS) [100] and the well-known eXtended Classifier System (XCS) from Wilson [259]. The basic concept is to create a set of rules that *classify* the current state, thereby creating a number of value predictions (similar to an expected reward) for each matched rule. This allows the system to choose an action, and apply the received reward to the rules responsible for it based on the original rule fitment values [242].

This has been used in the context of organic computing, for example by Tomforde *et al.* [237]. Instead of a fixed set of rules, a hybrid approach is created that distinguishes between situations where well-matching rules are already available, and more exploratory situations where new rules are created. This allows it to adapt in a more sophisticated way, instead of simply learning how to match an existing set of rules.

3.5.4.2. Neural Networks

Neural networks have dominated much of modern machine learning due to their adaptability to different scenarios. In basic terms they can be used to 'learn' any non-linear functionality [20]. A wide variety of different variants have been developed to fulfil different use cases, of which we will focus on two in this work. Firstly, the basic Artificial Neural Network (ANN) or Multilayer Perceptron (MLP) has been a staple for non-linear function estimation such as classification. Secondly, CNNs have gained massive popularity in recent years due to their ability to analyse both 1D and 2D input data – making them well-suited to both audio and images. A basic overview of how these two types of neural networks work will be provided here, focussing on the details of their computation.

Artificial Neural Networks The first neural network to consider is the classical ANN – or more specifically an MLP. These are the classical neural networks that have been popular in deep learning [84]. They consist of a number of neurons organised into layers as shown in Figure 3.6: the input layer, some number of hidden layers, and an output layer. Traditionally each of these layers would be fully connected with the next layer, leading to them popularly being referred to as Fully Connected (FC) layers.

The example in Figure 3.6 shows a network with 2 hidden layers being used as a classifier. There has been much argumentation in the literature about what makes a neural network a Deep Neural Network (DNN), but generally two hidden layers is considered to be the minimum.

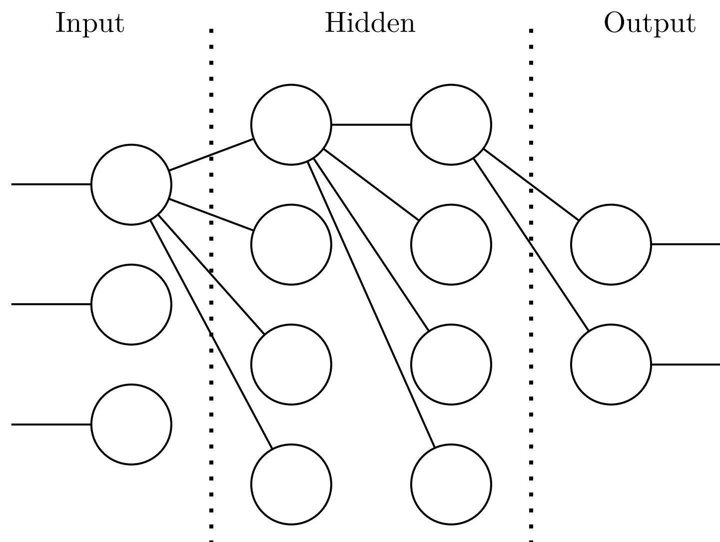


Figure 3.6.: Multilayer perceptron neural network structure

Using deeper neural networks has a number of disadvantages, for example overfitting

and unnecessarily complex computations. Overfitting is a common issue when training an excessive number of parameters for the training data [178]. Instead of generalising to the features that should be used to distinguish different training samples, it fixates on the specific training being used and simply 'learns' direct mappings for them. As is the case for many issues in deep learning, this balance requires considerable experience and gut feeling from the engineer creating the system.

The second issue mentioned relates to unnecessary computational load. Simply put, having more parameters and neurons in the network creates more processing work. When computing in the cloud this is less of a concern, but when developing for embedded systems it is crucial to optimise for performance and energy efficiency. Even when using hardware acceleration or high performance computing architectures, minimising the size of the network can be greatly beneficial due to the existence resource and energy limitations.

One recent development in neural networks to combat these issues is using a Shallow Neural Network (SNN) [154]. McDonnell *et al.* showed that using these shallowed networks that consist of only a single hidden layer (and using their extreme learning machine approach) can offer similar performance to deeper networks. Jiang and Crookes extended on this with their Small Unorganised Neural Networks and showed that for simpler problems they could outperform state-of-the-art DNNs [111]. By using a smaller network with fewer parameters, they aim to better generalise for less or simpler training data than larger counterparts.

Figure 3.7 shows how each neuron receives a weighted input from every neuron in the previous layer, and then creates a single output value through its activation function. Examples of such functions are ReLu, sigmoid, and step functions [84]. This allows the neuron to capture non-linear behaviour, creating a complex network of different non-linear functions connected together.

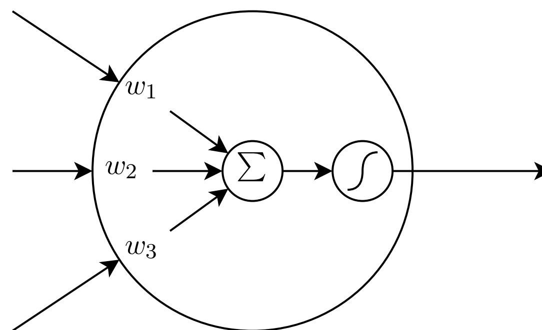


Figure 3.7.: Detail of single neuron behaviour, summing three inputs with weights w_i before going through a sigmoid activation function.

The complexity of computations required for a single neuron makes them very well suited to FPGAs and other hardware acceleration. Instead of each neuron requiring

multiple instructions to be computed (as is the case in basic sequential processors such as CPUs), the required arithmetic logic can be instantiated directly – allowing the entire neuron (or even more) to be computed at once. Since DNNs can scale up greatly in number of neurons (e.g. GPT-3 [29] has a ridiculous 175 billion parameters), this creates new opportunities for creating solutions that scale.

It is also worth noting that the network's feed-forward (general input to output computation) can be performed one layer at a time, since these basic neurons hold no memory and their output is not dependent on later layers. This greatly simplifies the parallelised implementation of such a network when compared to Residual Neural Networks (RNNs) [47]. Similarly, feedback (relating previous outputs to their inputs for learning) can be computed per layer, as is used in our ANN implementation for the FiPS project [75].

Convolutional Neural Networks The second variant discussed here is the CNN, which utilises many of the same concepts as the ANN. The important distinction with the CNN is that it performs the same operation over different sections of its input using its *convolution* kernel by moving it by a defined distance known as the *stride*. This allows it to consider both temporal and spacial dependencies in its input, as this information is persisted in the *features* computed by this operation.

A number of convolutional layers (known as its *depth*) may be followed by an FC layer. This allows the convolutional layers to identify interesting features in the input data, while the FC layer acts as classifier (similarly to an ANN). Commonly a pooling layer such as maxpool is placed between subsequent convolutional layers in order to effectively subsample the input, thereby finding larger features than is visible to a single kernel. Additionally, a batch normalisation layer may be present that serves to stabilise the input data by normalising the inputs for each layer.

The main computational complexity in CNNs comes from their large dimensionality. From the nature of a convolutional kernel, each element in the kernel needs to be computed individually before their outputs can be combined. This SIMD structure is why CNNs have been primarily deployed onto GPUs, utilising their parallelism to process large images.

3.5.4.3. Deep Learning Agent Policies

In practice, the number of states being considered in agent learning techniques such as Q-learning is non-trivial. This leads to a very high memory usage when individual explicit state-action values are stored. Therefore, neural networks have been employed to estimate the mapping from states to chosen actions in various different projects.

Deep Q Network (DQN) was developed by Mnih *et al.* [159], and trains a neural network using the same techniques as Q-learning. Different approaches have been created for mapping states to actions, either directly approximating the Q-function in Equation (3.2) or using an abstracted version of it. For example, Huang *et al.* uses the neural

network to generate a sequence of values that get converted to local/offloading allocations for a team of devices [105]. Ong *et al.* extended on this by creating a distributed variation that still relies on a server for computing the parameter update gradients [176].

An important use of deep learning to consider for this work is *DRL*, where different techniques are used to implement RL using neural networks. Generally, Deep Learning (DL) is used to learn and estimate the decision policy. Although deep RL implementations differ, one common factor is that a neural network is fed the current system state, so it can generate a new chosen action. Some specific implementations of this can be seen in Section 8.2.2 where we discuss a number of implementations.

Now that all of the fundamental concepts used in this work have been defined, we can proceed with the design of our solution. First, the three phases of our design as described in Section 1.2 will be presented: our hardware/software runtime in Chapter 4, optimised hardware accelerators in Chapter 5, and distributed autonomous learning of behaviour in Chapter 6.

Following the evaluation of all three phases in Chapter 7, we will present an overview of approaches followed by related work in Chapter 8. This serves to create context for our work, and augments the fundamental concepts presented here by showing alternative ways they could be used to solve similar problems. Although some projects have already been mentioned here in order to clarify certain concepts, a more complete overview of trends and the state of the art will be given there.

Chapter 4.

Elastic Node Platform

Modern problems require modern solutions, and the IoT is no different. Introducing increasingly complex AI into the system leads to cloud dependency, increased latency, or reduced battery life. Our hypothesis suggests that augmenting smart IoT devices with FPGAs provides opportunities for flexible, powerful devices that can provide local intelligence without sacrificing energy efficiency. This relies on a hardware platform that allows developers to fully utilise the local MCU and FPGA.

We introduce here our novel *Elastic Node platform*, which consists of a hardware platform and a software runtime. Its elasticity comes from including a local MCU and embedded FPGA, allowing it to adapt to changing requirements by deploying local hardware accelerators at runtime. Developing heterogeneous applications for this type of platform can be highly complex, and requires software support through abstractions and library support as covered in SREQ_{IV}: Convenient and Efficient Local Accelerators. At the heart of our approach to this is the *Elastic Node middleware* that provides a familiar interface for developers to use, and reduces work duplication.

As a whole, our runtime provides the foundation for creating smart IoT devices capable of local intelligence – along with supporting experiments in self-x research. Our goal is to give embedded software developers and distributed system developers (i.e. self-x and AI behavioural researchers) the tools they need to create real-world applications and experiments they need to evaluate their work.

We provide here information on the design and implementation of the Elastic Node platform starting with some motivation and background in Section 4.1, followed by the formalisation of a set of requirements in Section 4.2. After this, a high-level overview of the designed platform is provided in Section 4.3, and a deep dive of the abstractions offered by the stub-skeleton interface follows in Section 4.4. Some information about the hardware design follows in Section 4.5, and finally a summarising discussion is provided in Section 4.6.

4.1. Motivation and Background

FPGAs have seen noteworthy adoption recently in the automotive [174, 246, 252] and data centre [187, 218, 278] industries. This was accompanied by considerable support for developers – specifically with simplifying the creation of accelerator designs through

code generators and graphical system integration tools [153, 269] that reduce the effort required for connecting different design components. For example, the usage of standardised interfaces such as Peripheral Component Interconnect Express (PCIe) and bus systems such as AXI has reduced the complexity of local communicating in such platforms.

When we embarked on developing *FPGA*-enabled devices for autonomous systems in the IoT, however, we found that the world of embedded FPGA had not enjoyed the same level of simplification. Although a number of embedded FPGA-based platforms existed both COTS and in the academia, none of the available approaches satisfied our requirements (see Chapter 8 for a discussion on related solutions from literature).

Therefore, we decided to develop a platform specifically focussed on ease of use and targeting the lowest possible power envelope. Since we had experience in middleware solutions for distributed systems, the advantage they offer as a thin connection layer both between devices (or computational cores on a heterogeneous device) was clear. This initially manifested in a poster presented at ICAC 2017 [32], showing our initial design of having a minimal middleware layer to bridge the gap between the MCU and the FPGA. After receiving valuable feedback from the community, our path forward was clear: a hardware and software runtime for creating smart IoT devices that take full advantage of FPGA-based hardware acceleration — without requiring developers to entirely redevelop their existing hardware accelerators from scratch.

4.2. Platform Requirements

To ensure that our design satisfies our hypothesis and system requirements (see Section 2.4), we need to set a concrete set of requirements for the platform itself. These were originally based on requirements of similar projects in the literature [73, 208, 244, 258], and acted as a guide throughout the many design iterations of the Elastic Node platform. They were first published at ICAC 2019 [211], where we presented the third version of our platform. Following further discussions with members of the community (at the hand of a live demonstration), we were convinced that others shared our set of requirements.

We identified the following as the five core requirements for our platform:

$\mathrm{PREQ_{I}}$: Flexible Support for Real World Deployments

Firstly, it is vital that we can use the platform in real world experiments that take place in realistic IoT deployments. This means that the platform must support wireless communication so we can perform experiments relying on cooperating and offloading devices. Additionally, this drives our usage of energy-efficient components: to support mobile and battery-powered devices, the platform should be capable of lasting at least a few days on a single charge.

PREQ$_{\text{II}}$: Dynamic In-field Accelerator Reconfiguration and Control

Secondly, an application executed on our devices may require the usage of numerous different accelerators at runtime, as this is a major drawing factor of using FPGAs over ASICs. Our platform should facilitate this, by loading one of a set of configurations at will. Additionally, for periods of time the FPGA will not be required and should be turned off entirely to save energy, while the MCU runs the application.

PREQ$_{\text{III}}$: Easy and Fast Accelerator Access

Thirdly, utilising the accelerator within an application should be convenient and easy to do. By minimising the effort required, we aim to make the system more accessible to a wider audience and thereby improving adoption. To further encourage the usage of local hardware acceleration, we need to minimise the latency involved in switching between accelerators (or powering one up from a sleep state). If using these accelerators introduces huge delays in the application, users will be less inclined to take advantage of the possibilities they offer. In the same vein, accessing an accelerator as a coprocessor can involve considerable data being transferred between components. Therefore, we need to ensure that the platform provides state-of-the-art intercommunication rates between parts of the application executed on the FPGA and the MCU.

PREQ$_{\text{IV}}$: On-device Energy Measurements

Fourthly, the platform should act as a self-contained experimentation tool, and therefore should be able to report its own status (e.g. energy measurements). Instead of being constrained to a few devices that need to be physically connected to expensive monitoring equipment, a user should be able to deploy number of devices that are capable of monitoring themselves — without interfering with their ability to perform the application or communicate.

PREQ$_{\text{V}}$: Easy and Low-overhead Accelerator Reuse

Finally, such heterogeneous platforms often include considerable overhead in making an existing hardware accelerator compatible. We require convenient reuse of accelerators through a library approach – where existing accelerators can be easily incorporated into new applications, and new accelerators can be made compatible with our platform with minimal effort.

It is important to note that some of these requirements may be contradictory, such as ease-of-use and computational performance. When working with abstractions that

improve development convenience, balancing such requirements is essential. In many cases, compromises need to be made to satisfy all requirements.

4.3. Platform Overview

Each of the versions of the Elastic Node platform were slight iterations on the same fundamental core design. Our model relies on fairly tight coupling between parts of the application being executed on the MCU and the FPGA.

One important consideration is that we want the platform to be able to execute parts of the application on all available components simultaneously (see Figure 4.1). In this example, the MCU can capture the next data sample from a digital accelerometer while the previous sample is being processed by the FPGA. This improves overall efficiency since all components are fully utilised instead of having to wait on each other (thereby wasting time and energy). Alternatively, in some cases only some of the components may be required for extended periods of time (e.g. while the device is collecting large amounts of data to be processed later). Supporting both these variants allows our system to further adapt to changing workloads, growing and shrinking as the computational load changes.

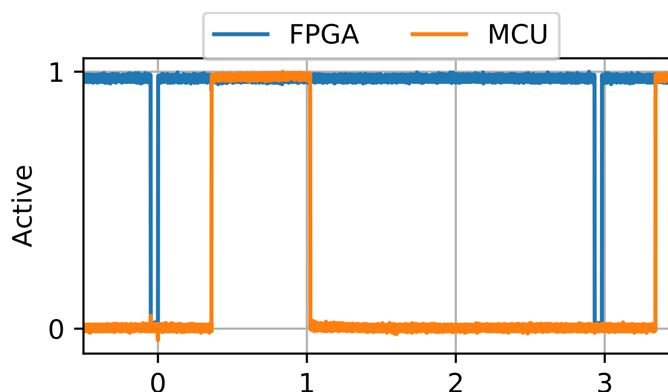


Figure 4.1.: Experiment showing simultaneous utilisation of MCU and FPGA in high performance mode, showing a neural network hardware accelerator (FPGA) that requires sensor data acquisition from a digital accelerometer (MCU).

Designing our own custom hardware platform provided us the opportunity to incorporate specific functionality that we required. One example of this is the self-contained energy monitoring required by PREQ_{IV}: On-device Energy Monitoring, which was not available when using existing COTS platforms. Additionally, having our own design with separate MCU and FPGA allows us to change out footprint-compatible components. This allowed us to have hardware platforms with different FPGAs on otherwise identical platforms – allowing direct comparative experiments.

At its core, the Elastic Node platform can be represented by the diagram shown in Figure 4.2, highlighting the MCU and the FPGA as cornerstone components of the system. It shows all of the abstraction layers separating the (1) Application and the (6) *Hardware Function (HWF)* – which is our term for a hardware accelerator that has been made compatible with our system.

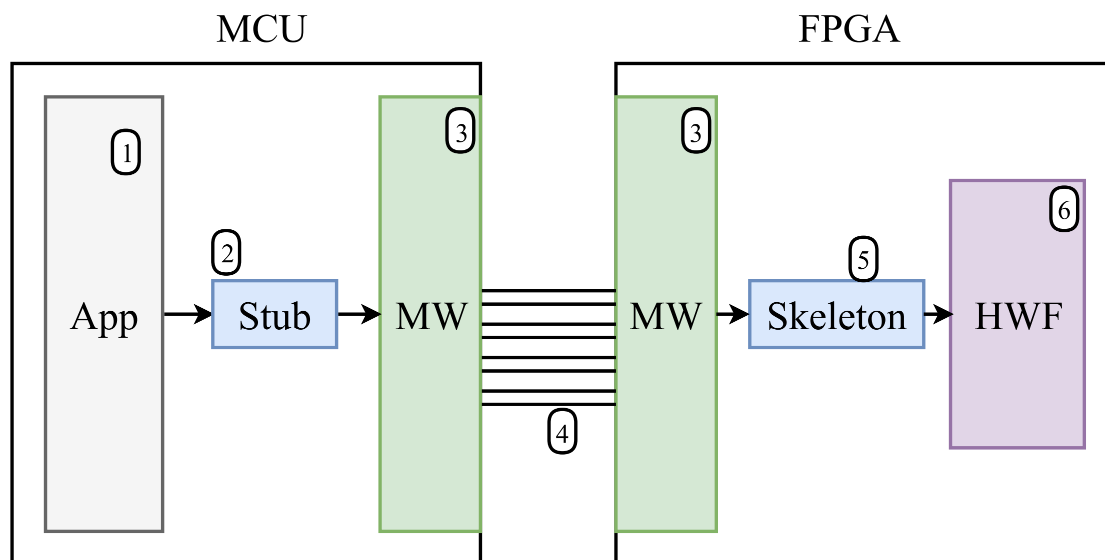


Figure 4.2.: Elastic Node system overview showing the important parts of the platform, from the Application on the MCU to the HWF on the FPGA

At the center of the design is the (3) middleware that spans the gap between the MCU and the FPGA, who are physically connected through the (4) interconnect. This middleware is then sandwiched between the (2) stub and the (5) skeleton, which are responsible for the data marshalling and synchronisation. All of these concepts will be discussed in detail in the following sections.

4.3.1. Hardware Functions

Before an accelerator can be integrated into our system, some minor alterations may be required. Once this is done, we refer to it as a hardware function because it offers hardware acceleration with the same simplicity as accessing a software function through an API. In most cases, this process is limited to identifying how it converts inputs to outputs (e.g. streaming data or more direct $A \rightarrow B$ computations) and standardising the data types used. More information on this is provided in Section 4.4.2, where all the requirements for an HWF are provided.

It involves defining and identifying each part of the interface as being *control* or *data*. The control interface includes some fundamental connections that almost every

design will have (e.g. clock, reset) as well as some more application-specific ones (e.g. handshaking, enable). The data lines can be either serial or parallel, but can also be extended into a standardised bus interface such as AXI [263].

The HWF can be defined using any HDL or other synthesizable language, giving the hardware accelerator developer the freedom of choice. Through the course of this work, we have developed and demonstrated a range of HWFs that provide the local device with increased application intelligence through highly efficient and fast computation. These include AI techniques such as a CNN [35], and more traditional ML ones such as Principle Component Analaysis (PCA) [36]. More details about these (and the optimisation process in general) is provided in Chapter 5.

4.3.2. Middleware

The central point of our solution is the middleware that is responsible for connecting the different parts of the system together. It is implemented partially on the MCU and partially on the FPGA, allowing it to bridge the gap between them. The main goal of our middleware is to hide the intricacies of developing on a heterogeneous system: communicating and synchronising between the different components. Its secondary goal is to provide a set of tools for controlling the system, further simplifying the development process by abstracting away complexities.

Our middleware is based on object-oriented middleware concepts popular in distributed computing and networking [97, 234]. This makes its core concepts familiar to a large group of developers, providing them with a familiar interface that can control embedded hardware accelerators.

One major benefit of using a middleware as abstraction is that it can be very thin without depending on complex OS functionality. This allows us to deploy it either standalone on bare metal platforms, or on top of an existing RTOS or embedded OS such as Linux. Standalone it creates the opportunity for deployment on severely memory-limited devices (addressing PREQ_I: Real World Deployments), as its memory footprint would be considerably smaller than that of a full OS.

Message-based systems are the main alternative, but are known to be very prescriptive of the software environment or OS used. They can also require substantial boiler-plate code for each application developed or each accelerator being used, as they offer lesser abstraction than a middleware. Instead, it relies on a protocol definition that describes each message being passed.

Even though the meticulous hardware-software co-development common in FPGA-augmented message-based platforms leads to highly optimised and efficient solutions (see Section 8.1.1), they require substantial application development effort since generally both the accelerators and application itself need to be written specifically for that platform. Our objective is to maximise code reuse both for the embedded software and the hardware accelerator developers.

The most important functionality provided by our middleware are

1. Software and hardware accelerator OTA updating,
2. Memory mapping and management, and
3. Accelerator deployment and management.

Firstly, to offer in-field reconfiguration and control one needs to ensure that both the relevant FPGA configuration and MCU program code are available on the device. Updating the MCU and the FPGA are surprisingly similar, relying on fragmenting the messages (due to message size limitations on the 802.15.4 wireless interface used in the Elastic Node hardware). Updating the MCU requires rebooting into a special bootloader [243] that can alter the program code stored on its flash memory. As the FPGA configurations are commonly loaded from an external flash chip, they can be written either directly by the MCU onto a shared flash storage or via a special FPGA configuration that provides a bridge from the MCU Serial Peripheral Interface (SPI) to the flash.

Next, the memory mapping of shared memory (between the MCU and FPGA) is used both to pass control messages and to provide access to the HWF itself. This creates a familiar interface for hardware-level embedded developers through specific memory locations, resulting in a similar abstraction to memory-mapped register-based access to peripherals found in bare metal programming with e.g. AVR and ARM microprocessors.

A diagram of this is shown in Figure 4.3, illustrating the two sections of memory as viewed from the MCU. The first is the internal memory containing local variables and other application data, while the second corresponds to the external memory that is accessed through the MCU-FPGA interconnect. This is further broken down by the middleware between Control and any loaded HWFs. Each of these can then be managed by the associated skeleton, as described in Section 4.4.2. Note that each of the stages in this is memory managed by the middleware, meaning that each layer can access its own memory from the 0x0000 address. This can be a convenient simplification when using precompiled libraries – especially in HDL.

Lastly, control of the deployed accelerator is provided through dynamic in-field FPGA reconfiguration (to satisfy PREQ_{II}: In-field Reconfiguration and Control). This can be done in a variety of different ways – dependent on the FPGA manufacturer as this functionality requires hardware support. All of the currently available versions of the Elastic Node hardware platform include either a Spartan 6 or Spartan 7 from Xilinx, which offer the same configuration options. However, our platform is compatible with any Static Random Access Memory (SRAM)-based FPGA, only requiring the ability to reconfigure in the field, and store newly received configurations for later use (e.g. on a local flash memory chip).

The types of reconfiguration that we currently support are

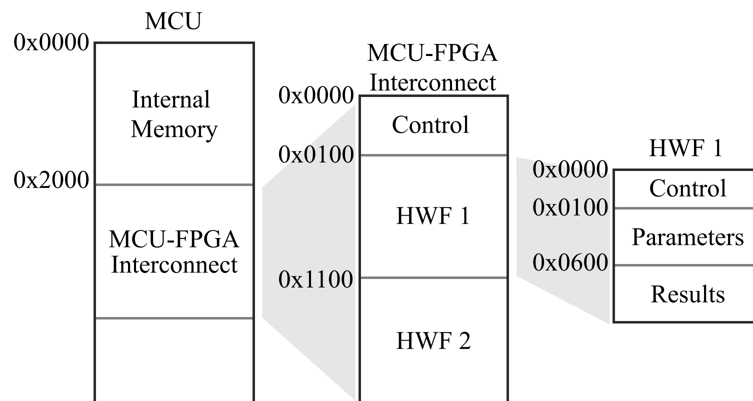


Figure 4.3.: Memory space management by the middleware

1. Flash master,
2. SelectMAP, and
3. Joint Test Action Group (JTAG).

The flash master directly reads the configuration from the flash memory to the FPGA configuration memory without requiring any other components. SelectMAP has the MCU push the new configuration to the FPGA in parallel (one or two bytes at a time depending on the implementation), while the JTAG method does the same using serial communication (and therefore requires fewer physical pins). We have experimented with the speed and convenience of these methods [98], and found that different options are optimal in different situations (see Section 7.1.2 in our evaluation).

Several other functionalities are offered by the middleware, and will be discussed as relevant. Since it primarily aims to provide easy access to accelerators and other system functionality, additional functionality can be integrated into it in the future.

4.3.3. Hardware-as-a-Service

While our middleware serves to make the local accelerator available to the local application, this does not make it accessible for other devices. To share processing power between peers and let them cooperate requires an additional layer of abstraction, which we refer to as Hardware-as-a-Service – referencing cloud nomenclature such as Software as a Service (SaaS) where services are centrally hosted and provided to clients. We describe here how the computational abilities provided by an HWF deployed to the FPGA of one Elastic Node can be shared with neighbouring peers.

As we presented in the Future Generations Computing Systems journal [33], the Elastic IoT Platform provides the ability to share any resource (e.g. a hardware accelerator deployed to an Elastic Node) with other parts of the application. All that is required is

for the device to describe the service provided (which may be as simple as identifying which accelerator is/can be deployed), and integrating the Remote Resource Framework (RRF) (which describes the created resource) into the Elastic Node middleware to process messages.

The platform user can access the provided resource as simply as accessing a Representational State Transfer (REST) interface — without being concerned with internal system parameters such as communication protocols. Additionally, a higher level placement service can be implemented using a digital twin that acts as a proxy that decides which request should go to which device. In this case, a computational service will be requested from the digital twin, and it transparently decides which device should be responsible for that functionality.

However, such a centralised service may be non-optimal as it does not scale with larger systems and introduces substantial communication overhead (some processing tasks and message handling that increase with more clients would need to go through a single bottleneck that creates additional overhead). Therefore, in Section 6 we design an RL-based system that can optimise this behaviour directly on the device. Currently offloading tasks from one device to another is done manually by the programmer, consisting mostly of sending the data passed through the middleware as a message. Integrating this message generating system into a code generator is left to future work.

4.4. Stub-Skeleton Abstractions

To simplify locally interacting with the intelligence provided by the HWF, a stub and a skeleton are used as abstraction layers. The skeleton is deployed onto the FPGA as a layer above the HWF, while the stub is deployed onto the MCU. This stub creates executable functions that an application on the MCU can call, simplifying interactions with the FPGA-accelerated functionality to a basic library that embedded software developers are familiar with. This creates an RPC-like interface, addressing our requirement for ease of reuse (PREQ_V: Accelerator Reuse) by reducing the integration of an accelerator to including the correct library.

Instead of this, we also considered a number of alternative approaches as introduced in Section 3.3. For example, one could use OS-level abstractions such as threads as used by ReconOS [143]. However, this requires a full OS to be deployed to the platform – which is impractical within the resource limitations when using tiny, cheap and power efficient MCUs. Another alternative would be using a modular tile design as used in the Erlanger Slot Machine [148]. However, this requires the design of special hardware accelerators which greatly increases the development effort required (compared to adapting an existing accelerator using a skeleton). Similarly, utilising the SoC approach where each system component interfaces through a shared bus increases both the resource overhead on the FPGA, and requires compatibility with that bus standard. That highlights our main motivations for using a middleware design that utilises a stub-skeleton abstraction: maximising compatibility with existing code while minimising the resource overhead on

both the FPGA and MCU.

Creating the abstractions of a stub and skeleton manually for each accelerator being used is tedious, however, and therefore a simpler approach is required to encourage adoption. We describe it here by first discussing the grammar of the IDL, followed by an overview of how local caching is performed for more complex interfaces, and concluding with a discussion on the offloading process from application to HWF.

4.4.1. Interface Description Language

The first step towards simplifying this process is to create an IDL that defines the interface created for each HWF's stub and skeleton. This describes the data and control interfaces, and identifies the type of skeleton (see Section 4.4.2). The main objective of this IDL is to create a common language that is compatible both with the programming language of the stub (e.g. C) and the skeleton (e.g. VHDL). It places the onus for integrating an HWF with the platform on the hardware accelerator developer, since they have the best domain knowledge for how it needs to be connected.

The IDL created for the Elastic Node platform is accompanied by a number of generators [260] for creating the relevant stubs and skeletons. Once the description is created either using the simple Graphical User Interface (GUI) or directly in text, a simple script can be used to create a C code stub and a VHDL skeleton. The VHDL file forms the top level of the implemented accelerator, as it includes the HWF and provides the needed caching and address mapping.

In simple terms, an IDL description of an HWF requires the definition of each port in the interface, along with some basic configuration parameters for the interconnect between the MCU and the FPGA. A simple example is shown in Listing 4.1, which demonstrates the required IDL for calculating $A + B = C$ where all values are unsigned 8-bit integers (`u8`).

The basic segments of an IDL are shown to be the `mcu` description (which describes the MCU-side interface), the `function` segment describing the HDL fundamentals (such as the library of the hardware accelerator and the name of the HDL implementation required), followed by the list of interfaces. In Listing 4.1 the *data* interface consists only of the unsigned 8-bit variables (`u8`) A, B, and C. The rest of the interface consists of *control*, which depends on the type of skeleton interface (in this case *oneshot*). This process is discussed at length in Section 4.4.2, and dictates a number of the choices shown here.

Although this listing might appear to be a long description for a very simple HWF, some of the fields shown can be omitted if default values are acceptable. These and a comprehensive description of our IDL is given in Appendix A. This includes its Extended Backus form[1], as well as various examples for some of the HWFs developed in our work.

[1] `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form` (last visited: 2021-11-27)

Code Listing 4.1.: Simple example IDL description for sum of two numbers

```
configuration BasicSum_example:
   mcu:
      wordsize = 8
      addresswidth = 16
      endianness = little
      activestate = high

   function BasicSum:
      hdl = vhdl
      endianness = little
      activestate = high
      type = oneshot
      library = work
      implementation = Behavioral

      bit clk -> clock
      bit reset -> reset
      bit start -> start
      bit learn -> ctrl_in
      u8 A -> data_in
      u8 B -> data_in
      bit dataRdy -> done
      u8 C -> data_out
```

4.4.2. Skeleton Interface Definition

An HWF's interface consists of two things: its skeleton type and data interface. The data interface is the simpler of these two, comprising mostly of what data is being exchanged with the HWF: each being a combination of data type and size. As the generated memory locations (see Figure 4.3) are hard-coded in the current implementation of the Elastic Node runtime, only fixed sized data inputs and outputs are supported.

More specifically, the fixed size data types used in C (all integer types from *stdint.h*) are used in our IDL for convenience, as this standardises the type definitions for both the MCU and the FPGA. For example, an unsigned integer with 8 bits is defined in C with the `uint8_t` type, while in VHDL we defined an alias with

Code Listing 4.2.: Standard *stdint.h*-style data types defined in VHDL

```
1 subtype uint8_t is unsigned(7 downto 0);
```

and similar ones for common integer types. Non-standard data sizes (not a power of two) need to be rounded up to the next available size, with the unused data bits optimised away by the compiler.

Alternatively, a bus interface can be used as is popular in multiple communities such as SoC developers. This can also be made compatible with our skeleton system, as most of the data management is managed by the bus system itself. Currently our system supports the Wishbone bus [177], which is an open source standard that supports various different data widths (with handshaking). We chose this bus standard for our initial implementation as it is entirely open source, compatible with various FPGA from different manufacturers, and its very light-weight implementation (in contrast to alternative standards such as the AXI bus from ARM [10] and Xilinx, or the Avalon bus from Intel/Altera [107]). This bus interface is not yet incorporated into our code generator, as this requires a more advanced system-design GUI.

Note that since any arbitrary complex data types (such as images) can be represented by a byte array, the standard types shown can represent any arbitrary fixed size types. Since these correspond to static arrays in C code, the sizes of these data types usually need to be predefined. Alternatively, VHDL types that do not have corresponding types in *stdint.h* (e.g. `STD_LOGIC` which refers to a single binary value) can also be used. For example, any logical flags in the FPGA interface are commonly mapped to one bit in a `uint8_t` (similarly to how AVR control registers comprise of up to 8 flags). Since binary values can directly be represented in hardware as a single bit, it reduces the extra mapped memory from defining each as its own full `uint8_t` as is commonly done in 8-bit software (or larger values for wider architectures).

Apart from the data interface, the control interface (and corresponding skeleton type) needs to be defined in the IDL. We streamline this in the current solution by defining three main types of skeletons that each describe a set of hardware accelerators:

1. One-shot,
2. Asynchronous, and
3. Streaming.

Although not a complete list of different interaction patterns, each leads to a different type of stub. They represent various combinations of blocking and non-blocking interactions possible in a sequential processor. They utilise synchronising with status lines, software interrupts, and buffering in the skeleton itself.

4.4.2.1. One-shot

This is the simplest interaction, and represents a blocking function call in the RPC interface. It requires only a logical `start` and `done` line in the skeleton, which control the beginning of the computation when all input data is available, and indicates when the output is ready to be read out. This creates a direct call that converts one set of inputs to one set of outputs. For example, direct mathematical operations such as a fixed size matrix multiplication ($matrix_c = matrix_a \times matrix_b$) utilises this type of interaction, as both inputs (the matrices) are fed to the HWF before the computation starts (shown in Listing 4.3). Once the computation is complete, the result can be retrieved from the skeleton and is returned to the embedded software developer's RPC-style function call.

Code Listing 4.3.: Example of one-shot stub function

```
1  void matrix_multiplication_15x15(uint8_t *matrix_a, uint8_t *matrix_b,
   ↪  uint8_t *matrix_c);
```

4.4.2.2. Asynchronous

This type of interface adds the ability to return delayed results, in a similar way to a non-blocking software function. It adds the `output ready` line that can is polled by the skeleton to indicate any available output values (plus a similar line for the input). This type of HWF may have both a variable *duration* and *number of outputs* based on the inputs provided. It is specifically useful for more complex operations such as Point of Interest (POI) extraction in CV, where an image is provided and any number of POIs are found. At any point during the processing of this image a new output can be created, which to the software developer appears as a function *callback*. This is shown in Listing 4.4 where an image of size $x \times y$ is provided, along with a callback that accepts a pointer to a created POI. This also allows the MCU to continue with another task (or save power by going into a sleep state) while the FPGA finishes processing.

Code Listing 4.4.: Example of asynchronous stub function

```
1  void poiExtraction(uint8_t *image, uint16_t height, uint16_t width,
   ↪  (void)(*poiCallback)(uint8_t *));
```

4.4.2.3. Streaming

Lastly, the *streaming* interface provides the ability to continuously push data into a pipelined accelerator, regularly creating output on each provided input. This can be used

as shown in Listing 4.5 for a Finite Impulse Response (FIR) filter, where every input value creates a new output value. Since this is also a blocking call, some computational latency may be introduced by the accelerator, and therefore it also has a very similar interface to the one-shot skeleton. However, the major difference with the streaming interface is that it expects to be called repeatedly internally while the one-shot expects only a single execution. Therefore, an array of n inputs are provided in the listing shown, creating n outputs and storing them at the output pointer.

Code Listing 4.5.: Example of streaming stub function

```
1 void fir_filter(float *inputs, float *outputs, uint16_t n);
```

Both the *asynchronous* and *streaming* interfaces highlight the need for caching inputs and outputs on the FPGA to somewhat decouple the interactions between the FPGA and MCU. Even in the case of the *one-shot* skeleton, one set of inputs and outputs need to be cached in the skeleton before being handed to the HWF. Therefore, we introduced input/output caching that can temporarily store data transparently in order to simplify the stub interactions.

4.4.3. Transparent I/O Caching

All interfaces automatically create a buffered cache in the skeleton that store inputs and outputs as required. This allows the FPGA and MCU to interface with the HWF at different rates, which can be very beneficial when a large clock disparity exists (the FPGA on our Elastic Node v4 operates at at least 4x the speed of the MCU). We identified the speed of the data interface to be the primary performance bottleneck during our original design of the Elastic Node platform [32].

Our solution to this is to use data caching, improving energy efficiency by allowing the MCU to continue transferring data while the FPGA starts processing. When using asynchronous or streaming skeleton types, this can greatly improve the overall performance of the system. This is illustrated in Figure 4.4, showing how the MCU and FPGA can be active simultaneously by caching inputs. While the first option needs to wait for the FPGA to finish processing and the output to be retrieved, having a cache store additional inputs and waiting outputs allows the system to better utilise each part of the pipeline.

The Elastic Node platform offers this functionality as part of the skeleton generation [260], where the user simply provides the depth of the cache (how many sets of inputs or outputs to store) in the IDL. This leads to a configurable data array being generated and instantiated on the FPGA, along with the required synchronisation logic for reading and writing each element. The software developer's interface to the HWF is not affected.

Through simple 'data ready' handshaking, the input variables are passed to the HWF

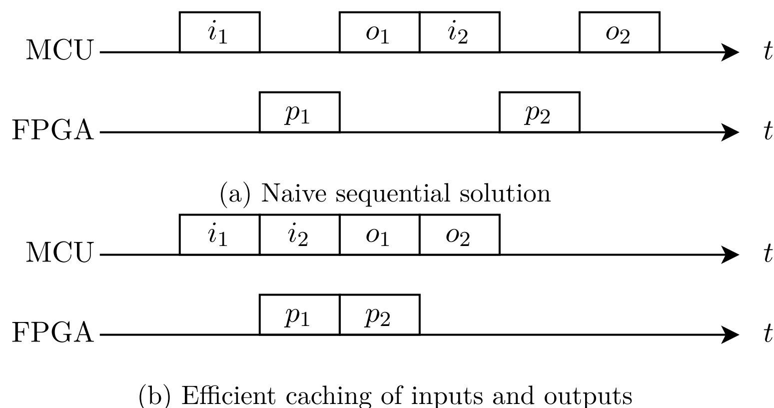


Figure 4.4.: Advantages of I/O caching, showing reduced FPGA downtime during continuous usage, since the MCU provides the second input (i_2) early enough to start immediately after processing the first sample (p_1).

as they are required and results are fetched as they are available. For example, creating a 40-element deep buffer for the output C from Listing 4.1 would be

```
buffer[40] u8 C -> data_out
```

which internally creates and subsequently hides the required handshaking lines required for loading and removing values.

4.4.4. MCU-FPGA Offloading Procedure

We refer to accessing the HWF functionality from the application layer as *offloading*, as the objective is generally to offload a computational load from the MCU to the computationally more powerful FPGA. This coprocessor-like interaction is done using an RPC abstraction, creating a virtual library that can be accessed by even a novice embedded firmware developer.

The process of offloading starts with the application calling the stub function, identifying which functionality is required and the parameters that should be passed to it (by way of the function parameter list). The stub is then responsible for marshalling the provided data into a generic data format that can be passed to the middleware. This bridges the gap between the MCU and FPGA, so that the data can be passed in turn to the skeleton. There it is demarshalled before being handed to the relevant HWF, along with any associated control signals. Fetching results from the HWF is the same process: indicating what data is required through its memory mapping and reading from that memory pointer.

An analysis of the timing of this process is shown in Figure 4.5, illustrating the time overhead introduced on the FPGA-side of the interaction. This graph defines t_0 as the point when the middleware samples the interconnect (the MCU asserts the data asynchronously so that is not included here). As soon as the address is sampled, the middleware decides if the interaction is control or HWF data, which takes one clock cycle (until t_1). The skeleton then needs to buffer the data for the HWF to use, which is another clock cycle until t_2.

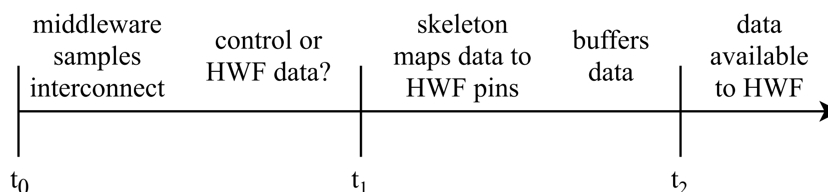


Figure 4.5.: Latency of offloading steps

All of this internal timing and functionality is hidden from the user, as they simply call the relevant function. By using a memory-mapped interface the MCU-side interaction is also simplified as much as possible, offering the highest speed interface available on the small MCUs we target.

4.5. Hardware Platform Design

Realising the design of the Elastic Node platform requires appropriate hardware as discussed in the platform requirements (see Section 4.2). This required us to create our own hardware platform that not only satisfies these requirements, but also allows us to easily change the components used for experimentation. Having our own design makes this much simpler than each time trying to find another appropriate COTS platform or modify existing hardware.

On a fairly high level, the hardware design can be represented as shown in Figure 4.2. It shows all the main components of the platform, as well as how they are connected. Arguably the most important components are the (1) MCU and the (2) FPGA. These are connected using the (6) interconnect, and each is connected to some (3 & 4) flash memory. Note that the flash memory can either be conceptually or physically separated, as some versions of the Elastic Node hardware have monolithic flash storage and others have separate flash modules.

The system also includes a (5) wireless communication module for interacting either with neighbouring peers or with a application/experiment server. This module is connected to the MCU, since it is more appropriate for the slow transfer rates of the wireless modules than the more power-hungry FPGA. Lastly, (7) energy monitoring logic is connected to each of the components – allowing it to separately monitor the usage of each part of the system.

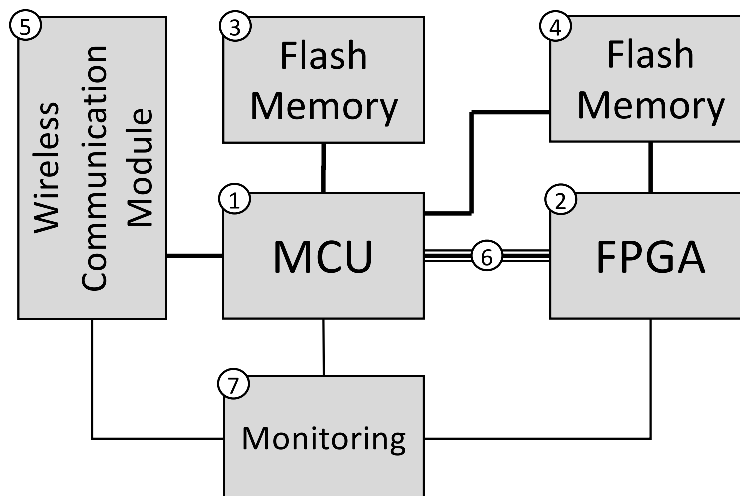


Figure 4.6.: Elastic Node hardware platform system overview

A number of these hardware platforms have been created through this work and accompanying research projects, as is shown in Appendix D. Each iteration of the Elastic Node hardware platform introduced a further improvement on the system design, or expanded on the supported hardware components – creating further opportunity for experimentation.

4.5.1. Hardware Interconnect

This interconnect is crucial for the responsiveness and performance of our system. In our work the MCU is almost always the communication bottleneck, simply because it is commonly clocked considerably lower than the FPGA (for maximum power efficiency typically 8MHz versus the 32-50MHz of the FPGA). Additionally, any communication interface needs to either be supported in hardware by the MCU manufacturer or needs to be emulated in software (which is again considerably slower).

Therefore, we experimented with the major communication interfaces available on our class of MCU, for example with the ATMega64 from Atmel. Note that we want to avoid interfaces that would be resource-intensive to implement on the FPGA, and therefore avoid Universal Serial Bus (USB) as this would require a full host implementation on the FPGA (as small MCUs such as the AT90USB128 used in other versions of the Elastic Node hardware platform support device but not host USB functionality). The results of our comparison is shown in Figure 4.7, showing the latency involved in transferring data chunks of various sizes. This is done to provide context for offloading different amounts of data during an interaction, as smaller interactions are less dependent on the interface rate. However, once larger amounts of data need to be transferred (e.g. images, point

clouds) this can cause large time (and therefore energy) overhead.

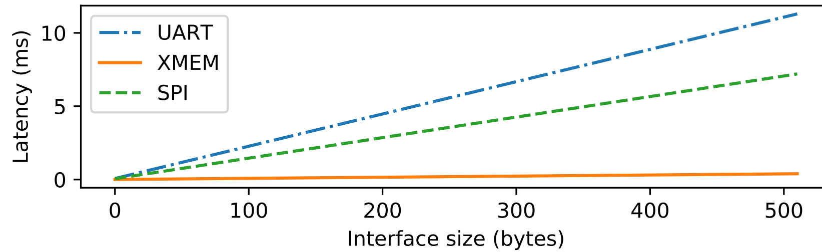


Figure 4.7.: Transfer speed of various communication technologies based on experimentation with transferring increasing amounts of data

The External Memory (XMEM) interface is shown to perform substantially faster than either Universal Asynchronous Receiver/Transmitter (UART) or SPI. This is because the XMEM interface on these devices is capable of transferring one byte (the width of the interface) fully addressed (up to 15 bits) every four clock cycles. This is faster than any alternative available on these devices, and requires no external hardware.

To evaluate the impact of the interconnect on the performance of the system, we measured the breakdown of an offloading operation. The time required by the MCU to prepare the data, the communication time, and the overhead introduced on the FPGA are measured locally on the Elastic Node as shown in Figure 4.8.

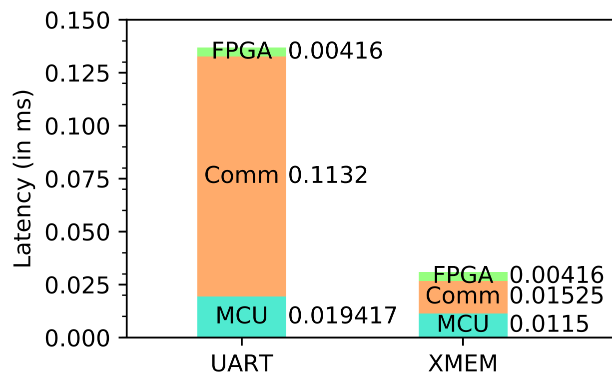


Figure 4.8.: Timing breakdown of MCU-FPGA offloading measured by the MCU during experimentation (averaged over multiple repetitions)

This shows the huge advantage of using this optimised interconnect instead of UART (as was used in the first versions of the Elastic Node hardware). Although the FPGA time is independent on the interface used, it is interesting to note that the time overhead on the MCU increases for the UART over XMEM. This is because sending via UART

introduces an initial delay as the data needs to be copied into the correct location, while the XMEM can directly copy it using the provided manufacturer library that provides `memcpy`.

4.5.2. Power Monitoring

Another important requirement for our platform was that it should provide on-device convenient power monitoring as prescribed by PREQ_{IV}: On-device Energy Monitoring. This requires more than just local monitoring of the overall power consumed by the board, also demanding more detailed insight into each component's consumption. This provides feedback not just for how well the device as an overall system is functioning, but provides detailed analysis that developers can use for future optimisations.

Additionally, local online power monitoring is a prerequisite we set for self-aware embedded systems. The self-aware controller needs access to this information in order to optimise the device's behaviour. This also means that the information needs to be easily accessible in the right format, requiring for a convenient API and library that provides access to the user as well as internal system calls.

4.5.2.1. Power Monitoring Hardware

Local power monitoring is introduced to the Elastic Node hardware by adding a number of current monitoring sensors. Current versions utilise the PAC1720[2] sensors, which can provide current (or indeed power) monitoring for two independent channels at up to 40 Hz at 11bit accuracy. By adding four of them to the Elastic Node v4, we can simultaneously monitor the various voltage rails of the FPGA, the MCU, the wireless transceiver, the monitoring logic itself, and the USB. Feeding all of this information about how much power each of these components are using back into the system provides opportunities for detailed experimental results as well as design space exploration and optimisation. Systems that aim to evaluate the performance of various system configurations can then simply deploy them each in turn, and compare how each component in the system is performing for each configuration.

These types of current sensors work by utilising a so-called *shunt resistor* as shown in Figure 4.9, where a low value resistor is added in series with the load that should be monitored (e.g. the wireless transceiver). The voltage over the shunt V_s is then monitored by the current sensor (with a known gain to improve digitising accuracy), as well as the source voltage V_{src}.

Combined with the known shunt resistance R_s (remember the load resistance R_L is unknown), the relationship

$$I = \frac{V_s}{R_s} = \frac{V_L}{R_L} \tag{4.1}$$

[2] `https://www.microchip.com/en-us/product/PAC1720` (last visited: 2021-12-13)

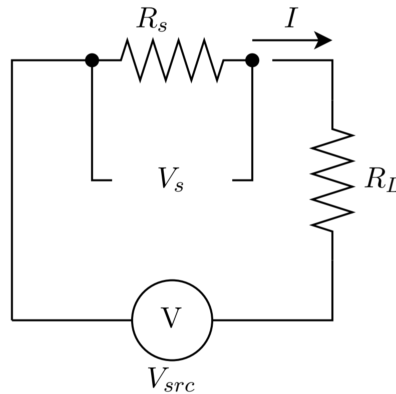


Figure 4.9.: Current monitoring via a shunt resistor

can be used to get the load power via

$$P_L = V_L I = \frac{(V_{src} - V_s)V_s}{R_s} \tag{4.2}$$

using only measurable information. Repeating this for each load to be monitored then creates a power monitoring system that can collect valuable performance data autonomously, with minimal impact on the overall system power.

In order to verify the accuracy of these measurements, we connected our oscilloscope (an R&S RTB2004[3]) over the shunt resistor to directly monitor V_s as a ground truth. After careful calibration, the results shown in Figure 4.10 were retrieved, showing that our measured data is exactly on the true value. We found that the sensors are accurate within their defined quantisation error (which is configurable from 6 to 11 bit data accuracy).

The current sensors are all connected to a shared Inter-Integrated Circuit (I^2C) bus, which is connected both to the main application MCU and the monitoring MCU. This second MCU is a lower powered model than the main application one, and does not offer some of the extended interfaces such as USB and XMEM. Its only role in the system is to regularly monitor the measurements from each of the sensors (which have very limited local storage available) and provide that information back to the main MCU upon request.

4.5.2.2. Monitoring Library

Accessing the raw power monitoring data is adequate for live demonstrations such as the ones we showed at PerCom 2018 [31] and ICAC 2019 [211]. However, it does not suffice for creating self-aware embedded systems that can improve themselves or alter their own

[3] `https://www.rohde-schwarz.com/us/products/test-and-measurement/oscilloscopes/rs-rtb2000-oscilloscope_63493-266306.html` (last visited: 0017-04-2022)

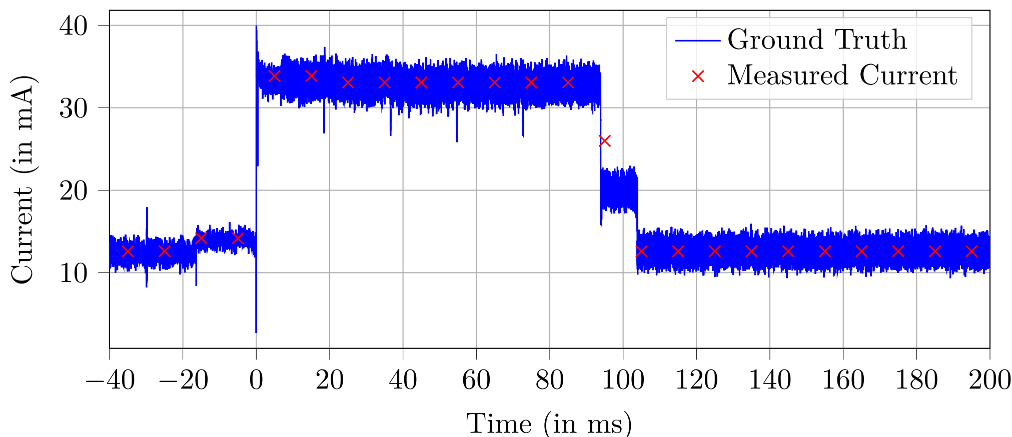


Figure 4.10.: Power monitoring verification, superimposing data captured by the power monitoring library (X) over the oscilloscope captured data (blue line) acting as ground truth

behaviour. For that, we need a more convenient API that retrieves the information in a more useful format.

This functionality is built into the middleware software [78], which provides a number of functions for retrieving the measured data. Since most of our systems are more interested in energy usage than instantaneous power, a Real-Time Clock (RTC) was introduced [213] that can accurately keep time across a longer space of time than most MCUs. This adds the functionality for easily starting and stopping an energy measurement, effectively creating an interface where the user can ask 'how much energy is component X using from t_0 to t_1'. By using only the created `start_measurement(component)` and `end_measurement(component)` functions, the user (or self-aware controller) retrieves the total energy used by the specified component.

4.5.2.3. Battery Level Monitoring

Apart from live monitoring of energy usage, current battery level can be a very valuable metric. Therefore, the energy daughterboard for the Elastic Node that we developed in cooperation with Schmidt [213] includes an analogue measurement of the current voltage provided by the battery. Combined with the maximum charge voltage and the minimum safe voltage (the lowest voltage it can be discharged to without being damaged), the current output voltage of the battery can be used to estimate its current charge level.

4.6. Discussion

The design of the Elastic Node platform was done as a direct result of not finding COTS or academic alternatives that satisfied our requirements. Since then, the industry has

produced a number of higher performant SoCs and individual FPGA and MCU components (e.g. RISC-V processors supporting custom integrated accelerators). Although offering higher computational performance than the components used in the Elastic Node hardware platform, these also do not offer the flexibility or convenience we required.

As set out in Section 2.4, two of our core system requirements for the device were SREQ_{III}: Energy Efficiency and SREQ_{IV}: Convenient and Efficient Local Accelerators. The convenience of deploying accelerators has already been partially evaluated through the provided use case examples and the description of the created abstraction layers – showing how the complexity is fully hidden from the developer behind a simple RPC. Along with the platform requirements provided in Section 4.2, the rest of these system requirements will be evaluated in Chapter 7.

At this point, the first phase of our project has been described and the Elastic Node platform has been presented as a reconfigurable IoT device. It is capable of performing real world experiments where both energy efficiency and computational complexity are critical. It is accompanied by the Elastic Node middleware that provides convenient access to device functionality such as accelerator deployment and power management, as well as a stub-skeleton abstraction for incorporating hardware accelerators into an embedded application.

The next step is to create adequately efficient hardware accelerators that fit into the resource limitations created by our emphasis on SREQ_{III}: Energy Efficiency. The resulting component choice of very small FPGAs and their limited PL makes it critical that deployed accelerators utilise the available resource as efficiently as possible. This is less common in traditional hardware architecture development for FPGAs that target desktop or server-grade hardware, where the primary objective is generally throughput and high clock rates. Therefore, we focus on the development of appropriate architectures for an FPGA-augmented embedded platform such as the Elastic Node.

Chapter 5.

Optimising Embedded AI Accelerator Design

Augmenting an embedded device with local intelligence is inevitably going to require more computations. In layman's terms: when we require the device to do more work, it needs more processing resources. Although having a local FPGA creates opportunities for introducing additional and flexible processing power, our requirements both for the system $SREQ_{III}$: (NF) High Energy Efficiency and to some extent the platform ($PREQ_{I}$: Flexible Support for Real World Deployments) outline a general goal of low energy usage.

Therefore, we dedicate this chapter to discussing how various types of local AI can be optimised for use in heterogeneous embedded systems that utilise hardware acceleration. The primary goal is to address Contribution 3: efficiently utilising the limited hardware resources provided by embedded FPGAs. To accomplish the novel level of local intelligence stipulated by our hypothesis requires optimisation at a design level, which includes both careful balancing of performance and low-power operation (achieved by choosing the right hardware components), and design changes to improve the effectivity and efficiency of accelerators (through cutting edge fine-grained design optimisations).

To accomplish this, we firstly discuss our general approach and design rationale in Section 5.1, followed by the introduction of some specific optimisation techniques for hardware architectures in Section 5.2. Then follows a number of case studies for specific AI accelerators that we have optimised for use with the Elastic Node in Section 5.3, where we describe both which optimisation techniques we used and how the overall design was affected.

5.1. Optimisation Approach

Due to $SREQ_{III}$: Energy Efficiency, the design of the Elastic Node platform presented in Chapter 4 focusses on small, resource-restricted FPGAs. This allows the devices to retain a small energy footprint, enabling them to be battery-powered (or use passive power sources such as solar or Energy Harvesting (EH)). This design decision has a cascading effect on the rest of the system. It changes which hardware communication interfaces, software development kits, and heterogeneous design techniques can be used.

It also jeopardises the benefit of utilising a soft microcontroller in the FPGA logic, since that annexes a substantial part of the available resources (the exact utilisation varies both with which FPGA is used, and whether optimising for area or performance [264]). One option would be to split an architecture into multiple stages of a pipeline, and deploy each of them in turn [52]. Although currently out of scope, this would be possible within our design.

Instead, our work primarily focusses on optimising a singular HWF deployed per device. The objective of this is to maximise available resources for the HWF without complicating the development of compatible accelerators. Rather than having to entirely redo the design of a designed architecture, an adaption layer (the skeleton) can be generated that sits between the middleware and the existing design. This should ease development for both the accelerator designer and the embedded software developer, without increasing resource overhead.

The optimisation approach can also then focus specifically on that single HWF, utilising existing knowledge within the FPGA accelerator design community. The techniques detailed below include both hyper detailed and macro approaches: either changing how tiny building blocks are instantiated and combined, or high-level changes to the types of computations being performed. This serves to illustrate the importance of optimisation at every level of the design, as the design can only be truly efficient when fully optimised.

The techniques presented here are not a complete guide to hardware accelerator optimisation, but provide an overview of the available options. It also shows the various different resource types that need to be considered: ranging from the different types of memory to the sophisticated DSP computing slices. It also covers some techniques for increasing the possible clock speed of the design, which is dependent on the design architecture (longer, more complex logic between clock synchronisations reduce maximum clock rates) and can considerably improve the energy efficiency.

5.2. Hardware Architecture Optimisations

Many years of desktop and data centre FPGA adoption have led to well developed hardware architecture design techniques. With code generators and high-level synthesis tools like Xilinx HLS [269] being available, these classes of devices offer relatively easy development environments. However, the smaller classes of devices such as the Spartan and Artix families from Xilinx commonly have limited support for such tools.

Additionally, few of these systems support the development of heterogeneous systems that incorporate various processing cores – with the exception of soft-core-focussed SoC systems like the Xilinx EDK [264]. Our work is based on a different approach where accelerator developers can utilise their existing optimisation skills. Providing them an easy way to interface conventional accelerator designs into a embedded heterogeneous application simplifies their task – without adding the FPGA resource overhead involved with using a soft core processor. However, this relies on the abilities of the accelerator developer to ensure that their design (a) fits into the available resources and (b) provides

adequate performance to achieve high energy efficiency.

As a demonstration of how a design's energy efficiency can be improved, we showed [191] that the increase in total power consumption of hardware accelerators can be minimal as the implemented clock speed increases. By using a variable clock source (a feature of the Elastic Node hardware platform shown in Figure D.5), we gradually increased a basic FIR filter's operating frequency and measured the power consumption at each frequency. The result can be seen in Figure 5.1. The power usage on some of the internal device voltages (V_{CCAUX}, V_{CCBRAM}) saw no increase in consumption as they power sections of the device that were not utilised, while V_{CCO} and V_{CCINT} saw a linear increase with higher speeds. Overall, it was shown that the FIR filter at the maximum of 32MHz used only 8.9% more power than the slowest version at 1MHz while increasing the performance by 32x.

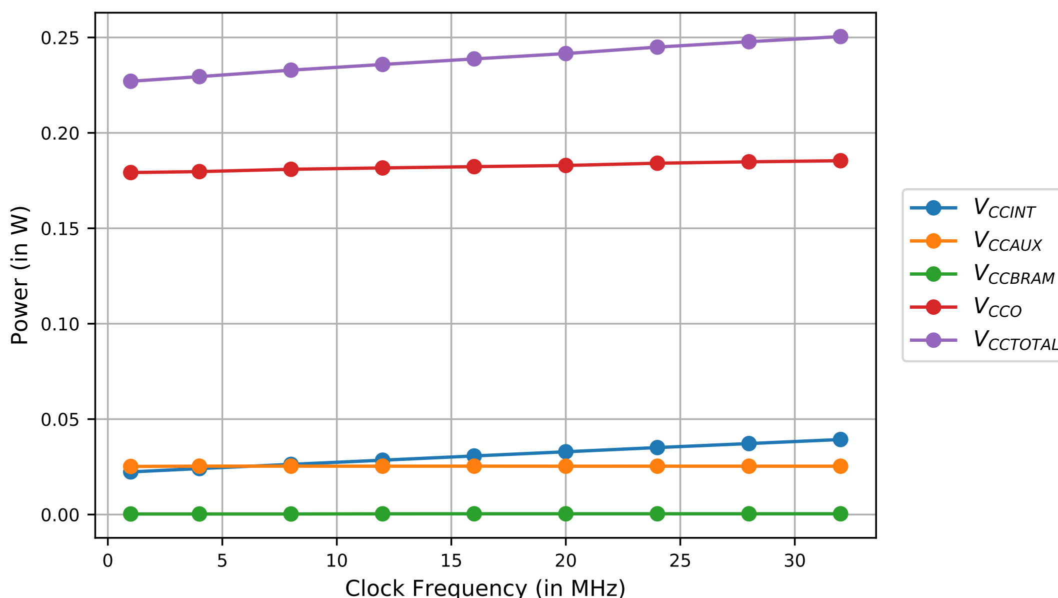


Figure 5.1.: Experimentally measured power usage on various voltage rails for FIR filter with increasing clock frequency, showing largely linearly increasing power usage for faster hardware accelerators

This illustrates that although the power usage of FPGAs increases linearly with higher clock speeds, a faster hardware accelerator will usually perform more efficiently overall. Over a batch of 1024 computations, this accelerator decreased in energy consumption from $22.7\mu J$ to $7.83nJ$ with higher clock rates due to the relatively high static power usage – which is independent of clock speed. A more extensive study of this follows in Section 5.3.

We present here some examples of hardware design optimisation techniques – some

well-known and others less common. Although not a complete list of optimisation techniques for hardware accelerator designs, the ones presented here proved to be the most beneficial for our example hardware accelerators presented in Section 5.3. They represent a wide variety of techniques, ranging from basic implementation optimisations to modern design developments that completely alter how an accelerator functions. One common theme is our objective to create parametrised accelerators that can be tailored to a developer's requirements (e.g. speed, accuracy, or energy efficiency).

5.2.1. DSP Timing Optimisations

One branch of optimisation technique from the hardware architecture design field involves improving the timing of mathematical operations performed using DSP elements. We present here an example of this which aims to reduce the resource consumption in complex sequential multiplications: creating a sequential circuit that reuses the same MAC module repeatedly. Normally the computation of

$$A = B \times C \times D \tag{5.1}$$

would require two multipliers in order to compute the value of A within a single clock cycle. By adding a register E, this can be simplified through

$$A = (B \times C) \times D = E \times D \tag{5.2}$$

which allows the synthesizer to reuse a single multiplier. It calculates E in the first clock cycle, and A in the second – improving the maximum system clock speed by simplifying the required computational logic within a single cycle.

Another technique used is *retiming*, where registers are moved across combinational logic [181]. Using fine-grained manual optimisations such as these allows the developer to improve the performance of the system without changing the inputs or outputs of a logical circuit.

5.2.2. Volatile and Non-Volatile Memory Tiers

Apart from functional resources like LUTs and DSP slices, the available local memory can be very limiting when designing for embedded FPGAs. Xilinx's Spartan 7 family range from 180kb to 4,3Mb total integrated BRAM, while the larger Kintex family sports up to 34Mb. This amount of memory can be limiting for embedded application developers, who generally avoid using external memory such as Synchronous Dynamic RAM (SDRAM) due to the additional energy usage and cost. Another concern is balancing the amount of functional resources and the memory requirements. For example, reducing the use of registers (instantiated with LUTs) can lead to more localised memory being required.

The variety of different memory technologies present on modern FPGAs creates a large design space for the memory configurations – requiring a detailed analysis of each chunk of memory used [191]. We considered in our design not only the quantity and volatility

of each type of memory, but also the different energy usage profile (combination of static and dynamic power usage) of each memory technology. Generally, higher memory usage will lead to greater energy costs, but exceptions exist such as block memory types (e.g. BRAM) being activated in large chunks – leading to step increasing energy usage. Some experiments that demonstrate this are shown in the evaluations in Section 7.1.1, comparing the energy consumption of different memory configurations for the same accelerator.

5.2.3. Minimising Expensive Operations

There are some operations that are very expensive to implement in hardware accelerators. One example of this is division, which is commonly implemented using loops when dividing by a constant or iterative techniques such as Newton Raphson [277] for fixed-point operations. Since this commonly involves multiplication with reciprocals (which are again very resource intensive), a common approach is to simply avoid performing divisions as much as possible.

Some designs can be altered by using a different (cheaper) approach. This may involve using a different algorithm (e.g. using a matrix multiplication to solve a common transformation), or an approximation (e.g. a Taylor series for solving trigonometric functions). This highlights an important concept in hardware acceleration: the minimum required accuracy. In some cases using simpler integer representations improve performance or resource consumption without sacrificing overall the computed results, which has been a popular optimisation done for GPUs [94].

5.2.4. Floating Point Representation

Sequential processors such as CPUs incorporate dedicated floating point operators in hardware that allow them to cheaply perform highly accurate computations. However, as hardware accelerators implement each usage of a mathematical operation with its own dedicated circuit, using lower accuracy numerical representations can be a valuable simplification. In fact, implementing floating point mathematical operations are traditionally very badly suited to FPGAs [74], which is why they strongly rely on *fixed point* representations.

There are two aspects to finding an optimal fixed point representation (often represented with $\mathbf{Q}m.f$): adequate fractional bits (f) to maintain computational accuracy, and enough integer bits (m) to avoid overflow. While the resulting accuracy for various numbers of fractional bits can be empirically established through simulation (such as shown for our CNN hardware accelerator in Section 7.2.1), the simplest way to find the minimum required integer bits is to study the range of values that need to be captured. By finding the greatest value x (along with the smallest negative value y) that needs to be represented, the optimal number of integer bits can be simply calculated using

$$m = log_2\left(max\left(x, abs(y)\right)\right) \tag{5.3}$$

which needs to be rounded up. This ensures that every value to be represented by the application will fit in the representation chosen. Estimating required accuracy is more complex, and requires an investigation of the resulting error of changing the Least Significant Bit (LSB) at various points in the algorithm (as discussed in Section 7.2.1).

5.2.5. Utilising LUTs for Precomputation

A powerful technique for optimising the performance of hardware accelerators is using precomputation. This avoids doing specific computations that are used frequently, instead storing precomputed results in a LUT. When enough resources are available to store all possible distinct answers, this can provide a faster and more resource efficient solution in some cases than computing them at runtime.

At a high level, an FPGA LUT is effectively a register that can be addressed using a number of input lines. In the case of the 7 series FPGAs from Xilinx [265], the LUTs can be used as either 6-input 1-output or 5-input 2-output. This effectively creates a 64-bit register where each bit can be individually chosen as the output in a Multiplexer (MUX)-like design. This can be expanded by connecting multiple LUTs together, thereby providing more input or output lines (commonly a combination of both).

5.2.6. Latency and Throughput Modelling

The scaling in power consumption and processing speed between hardware acceleration and traditional sequential processing is greatly different. Therefore, careful modelling can be very beneficial to get an estimate of expected energy costs and performance. Especially to novices, this can be a valuable initial step when creating a new application. Combined with the results of an implementation tool such as Vivado from Xilinx [267], this can provide a good estimate throughout the development process.

By utilising the latency introduced by each type of operation (LUT, DSP, and logic gate), a very accurate estimate can be achieved. This can be created either by studying the HDL, or from the output of the synthesis process. One critical data point provided by this process is the critical path, which provides the latency (and location) of the slowest clock-to-clock computation in the design. This directs the developer to the most important area for optimisation.

Alternatively, one can create a high-level model of the algorithm that creates a parametrised estimation of the latency/throughput. For example, a common FIR filter with n taps that uses a single MAC element requires n computations (or clock cycles) for propagating through the entire filter and combining the results. Similarly to the Big O notation of a piece of software, that provides the developer with a configurable performance metric: allowing them to optimise the depth of the filter with its latency.

5.3. Example Hardware Accelerators

As our objective is to support a variety of different types of intelligence on the Elastic Node platform, we discuss here a selection of AI and ML algorithms. We have adapted each of these to efficiently utilise the limited resources available on our embedded FPGAs using a combination of the optimisations discussed in Section 5.2.

The objective of each use case discussed is subtly different, since each aims to demonstrate a different aspect of adapting a hardware accelerator to the Elastic Node runtime. This will begin with the interfacing logic and skeleton design for an ANN, followed by a study of efficiency optimisations in a CNN, and lastly we will discuss how larger problems can be made to fit limited FPGA resources available with IPCA. Each of these will also include a brief technical overview of the accelerator implemented, as well as which optimisations were used during its development.

5.3.1. Artificial Neural Networks

The first case study involves an ANN, or more specifically an MLP as discussed in Section 3.5.4.2. The main objective of this case study was to investigate the feasibility of local AI computation using an FPGA. This was demonstrated to the pervasive computing community at PerCom 2018 [31] by comparing it to wireless offloading.

This demo involved processing a batch of input data using an ANN locally on the embedded FPGA, and comparing its energy consumption and latency with offloading the computations to a nearby computer (representing a best case server scenario with no internet latency). Instead of transmitting the neural network's input data and waiting for the result, it offloads to a local FPGA.

Its primary goal was to show that using a heterogeneous embedded platform such as the Elastic Node was a feasible alternative to traditional offloading. In certain situations (e.g. when using larger batches to minimise overhead), it used less power than sending the work via 802.15.4. Due to its 50 MHz processing clock, it showed great promise for processing ANNs locally on an embedded device – capable of processing even networks consisting of hundreds of neurons within microseconds [31].

It also served to introduce the Elastic Node hardware platform to the community, and to receive valuable feedback about their requirements and interests when it comes to a hardware platform for experimentation in smart IoT devices. Although they were impressed by the low energy footprint of the platform (operating at 80.1mW during active usage) there was a demand for greater computational power and more complex accelerators. Along with high importance placed on ease of development, this was considered during development of later iterations of the Elastic Node hardware platform.

The VHDL-based design used to compute the ANN was adapted from an architecture we developed for the FiPS EU Project [75]. However, that design was targeted at a considerably larger FPGA with more resources (such as the Xilinx Zynq XC7Z045 that incorporates a Kintex 7 with 350k logic cells). Therefore it needed to be adapted to fit

within the resource constraints of the Xilinx Spartan 6 LX9 (which sports 9k logic cells) used in that demonstration.

Parallelisation Options The ANN HWF adapted from the FiPS project originally instantiated the full neural network in FPGA logic. This meant that each layer could operate simultaneously, meaning that both feed forward and feedback were possible at one layer per clock cycle. This leads to very fast computation of a full ANN as they are commonly much wider than they are deep – even deep learning models are commonly a few layers deep in order to simplify their training [84]. Although highly performance optimised, this design was much too big to be implemented on an embedded FPGA [31].

The solution was essentially that instead of instantiating the full neural network, we only instantiated a single neuron in the reconfigurable logic of the FPGA. While this scaled down the performance from the original designed aimed at server-based FPGAs, this was unavoidable as the training and inference of a neuron requires a considerable number of different computations.

For a network of depth d and width n, processing a single set of input data was increased from d to $d \times n$ clock cycles. While this matches the typical $O(dn)$ complexity for CPU-based solutions, they require multiple distinct instructions per neuron (largely multiplications of weights and connection values). Therefore, a hardware accelerated design that can reduce each neuron to a single clock cycle has an inherent advantage.

Optimised Volatile and Non-Volatile Memory Tiers The original FiPS ANN model offered enough registers on the FPGA to keep all required weights, biases, and computed connections in memory. This was fairly memory intensive, and therefore we altered this so each neuron's weights and other variables are loaded in turn from local BRAM.

This required complex memory synchronisation and addressing logic, which greatly increases the the complexity of the design when compared to a purely register-based one. Most FPGAs (e.g. the Spartan 6 and 7 families) use a multi-tier memory design, as discussed in Section 5.2.2. This is similar to the caching policy of modern desktop CPUs, where memory is organised into nearby L1 to L3 caches and secondary memory in the form of Double Data Rate (DDR) Dynamic Random Access Memory (DRAM). On an FPGA, this takes the form of distributed local registers, regional BRAM, and external SRAM or DRAM modules.

The basic approach to allocating each memory tier is to use the lowest level that has enough memory available. Therefore, the memory required for the ANN used for the demonstration in PerCom 2018 was allocated in the BRAM. It is important to note, however, that although memory size increases with each tier, the interfacing logic increases in complexity. For example, local registers require no addressing or timing and are always available, while BRAM needs explicit read/write logic and addressing.

Skeleton Development As the ANN accelerator was one of the earliest ones we developed for use with the Elastic Node runtime, it was used to develop an early version of

the skeleton system. This required the manual creation of the needed logic to demarshall the control commands and data coming from the application/stub. This was very valuable when creating the skeleton system, offering an example of a one-shot skeleton without need for data caching.

The adaptation of the ANN hardware accelerator – although the simplest use case presented here – was crucial for the development of the Elastic Node runtime. It was used both as a proof of concept for deploying complex AI to a heterogeneous embedded platform and during development as a blueprint for skeleton definition and generator development. Therefore, we will use it to evaluate the ease of development – specifically the creation of an accelerator's IDL description – in Chapter 7.

5.3.2. Convolutional Neural Networks

Another popular form of neural networks in the last years has been CNNs, particularly due to their ability to handle image processing in their 2D form [84]. Even in the simpler 1D form they have been shown to be highly effective for NLP [82], or even medical applications such as an Electrocardiogram (ECG) [35]. On a high level they work by performing a convolution over input data [84] – similarly to how image processing kernels are used in computer vision [149].

Due to a CNN's increased complexity over a traditional ANN, our primary objective for this use case was to ensure high energy efficiency without compromising ease of development. Using the Elastic Node v4 (sporting a Spartan 7 instead of the Spartan 6 used in the v3), we demonstrated this efficiency during a live demonstration at ICAC 2019 [211] by showing detailed graphs of the power usage in real time.

This was made possible by the accelerator design we created in cooperation with Qian [191], who investigated the impacts of different possible optimisations on the resource and energy consumption. The results of this were presented at the Pervasive IoT workshop at PerCom 2020 [35]. They are also provided in the Evaluations chapter, specifically in Section 7.2.2 which considers the effects of various common hardware design techniques presented here.

The main objective was to show how easily highly efficient AI accelerators can be created by converting a 'normal' TensorFlow [85] model. While the skill set required for designing a conventional CNN model has become fairly common, implementing one on resource limited hardware is considerably more complex.

To start with, we created a model in TensorFlow capable of high precision and recall – as shown in Figure 5.2. This illustrates that each of the 6 classes could be accurately identified reliably, achieving 97% accuracy overall. This model provides a good starting point, but since it relies on a number of incompatibilities such as floating point computations (see Section 5.2.4) it cannot be efficiently implemented as-is in hardware.

We address this by using a number of hardware accelerator optimisation techniques and some application-specific simplifications. These simplifications include moving the

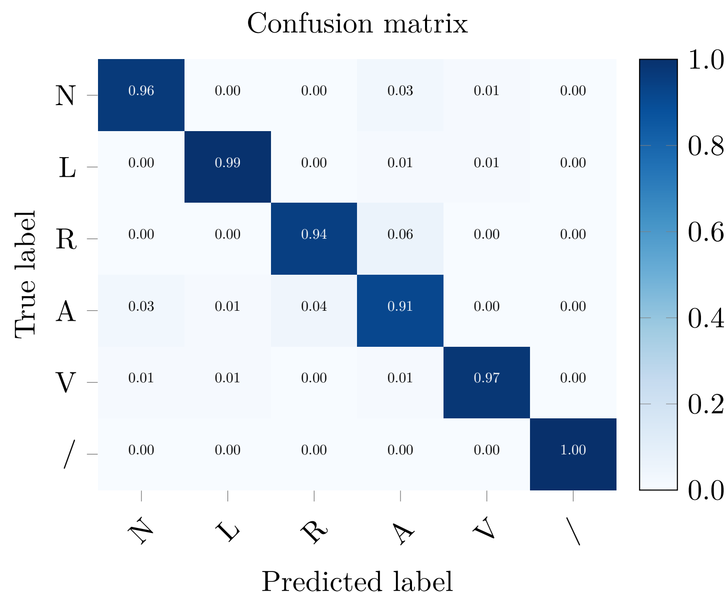


Figure 5.2.: Confusion matrix showing performance of original TensorFlow model, showing that each class can correctly be identified with an accuracy of 91% to 100%. Based on computed label predictions for the test set (33%) of the MIT-BIH Arrhythmia Database [162]

complex CNN training to the cloud to only compute inference locally on the device – under the assumption that these networks are often not dependent on online training. This further optimised communication with the device, reducing it to occasional model updates and reporting low-confidence classifications [35].

Inference Latency Modelling Another important contribution of our CNN hardware accelerator is the model for estimating system performance and accuracy. This includes a clock-cycle accurate estimate of the full network processing time. Importantly, this model is fully parametrised for which model is being implemented. After retrieving the maximum clock speed from the synthesizer software, the user can enter the model parameters (e.g. the number of layers, the number of features per layer, etc.) and compute an accurate estimate of the overall performance.

As an example, consider the following latency model of the CNN hardware accelerator that we developed for ECG processing [35]. It serves to make our CNN HWF adaptable to user requirements, creating an important balance between its computational accuracy and resource consumption. Please see Section 3.5.4.2 for the technical overview and a description of the parameters used.

The overall goal is to compute the latency of the full network

$$t_{CNN} = t_{clock} \times n_{total} = t_{clock} \times \left(\sum_{i=1}^{N} n_{c_i} + n_{ga} + n_{fc} \right) \tag{5.4}$$

where n_{total} is the total number of operations required. Similarly, the number of operations for each of the N convolutional layers is given by n_{c_i}, while n_{ga} and n_{fc} describe the operations for the global averaging and fully connected layers in turn.

It is important to note here that the meaning of an "operation" depends greatly on the design of the architecture. Using a general purpose CPU, this would be considerably higher than with our tailor-made HWF – since custom operations can be instantiated for computing nearly arbitrarily complex procedures. Through this, hardware acceleration can surpass the performance of general purpose computing — even when operating at a considerably lower clock frequency [83, 149].

Next, each of these numbers of operations n need to be computed: starting with the convolutional layers. By using

$$n_{c_i} = f \times K \times S \tag{5.5}$$

with f is the number of features being computed, K is the kernel size, and S is the convolutional steps. Since we use a stride of 1, the number of steps S can simply be computed through

$$S = H - K + 1 \tag{5.6}$$

by using the input height H. The input height of every subsequent layer is then halved through the built-in max pool operation.

Similarly, we can compute the number of operations for the final global averaging layer to be

$$n_{gc} = f \times (H_{ga} + 1) \tag{5.7}$$

again using its height H. All that remains is then the fully connected layer, which can be simply retrieved through

$$n_{fc} = f \times C \tag{5.8}$$

with C being the number of classes classified. Note that like many of the other layers, this layer could be reduced to fewer (more complex) operations. This would, however, drastically increase the resource requirements for instantiating it in hardware on the FPGA.

Using this model, we can calculate the operations required for the CNN used in our ECG project [35]. Since it classifies $C = 6$ classes using $f = 18$ features, it leads to a total of $n_{total} = 143262$ operations. To compute the real time latency of an inference run, the clock speed needs to be specified. This will be done (along with real world verifications) in the Evaluation in Section 7.2.3.

Dealing With Noise A common concern with using CNNs is data noise, commonly caused by low-voltage sensors such as ECG [35, 193, 227]. We found that this is especially problematic with small, power-efficient sensors used in embedded environments [103] as they offer low recording quality and low Signal-to-Noise Ratios (SNRs) (the relationship between the levels of the measured signal and noise). That makes input preprocessing and filtering paramount when designing hardware accelerators.

Therefore, we compared different options to preprocessing for CNNs [67], finding that different options are appropriate to each application scenario. For example, we found that the Savitzky-Golay filter (that smooths a signal using convolution and fitting a polynomial function to each subset of provided data) performed well when classifying urban sounds, but that the Discrete Wavelet Transform (DWT) was able to better process ECG signals with significant baseline wander when combined with FIR filters.

Optimised Volatile and Non-Volatile Memory Tiers When designing our CNN hardware accelerator [35], the weights and biases had to be loaded from local memory in a similar fashion to the ANN from Section 5.3.1. Since these do not change during normal querying (unlike our ANN design which can be trained online), they are non-volatile.

Since the volatile data (such as intermediate values of layer connections) constantly gets rewritten, we need to consider both reading and writing them to memory. The dual-port BRAM present on modern Xilinx FPGAs [270] offered the advantage of being able to read one location in memory while another is written – e.g. output connections of one layer is written while the next layer's data is preloaded for computation in the next clock cycle.

Additionally, some static parameters are stored in LUTs used as registers. These offer very fast onboard memory, and are located throughout in the PL – leading to simpler (and therefore faster) routing when compared to BRAM or external memory.

Utilising LUTs for Precomputation One application area for on-device CNNs is in medical wearable ECG sensors. A modified version of our CNN architecture was used in the LUTNet project [70] for finding heart rate abnormalities. By using advanced techniques like Binary Neural Network (BNN) and precomputing using LUTs (see Section 5.2.5), a tailor-made system was created that combines local energy efficiency and high performance.

In this project, convolutional layers are replaced with LUTs for increased efficiency in embedded AI applications. The basic concept is to pre-compute a layer into a LUT on an FPGA, simplifying the numerous DSP-based computations required into a single lookup.

Modular Layer Combination Implementing subsequent convolution, max pooling, and activation layers independently (as is done in sequential processing models) requires the storage of intermediate values. Instead, our hardware accelerator design combines these

layers into a single pipelined computation – thereby avoiding the mentioned additional complexity.

This is possible due to the relatively low computational complexity of the max pooling and activation layers when compared to the convolution layer. A pipeline was created that allows the accelerator to perform these three procedures within two clock cycles. Instead of saving each intermediary result in memory, they can be pushed through by using a shift register implemented using LUTs. Apart from reducing the amount of memory required, this also removes the need for complex addressing logic for loading and storing to the memory (as is present on our ANN).

Exponential Functions The last step in a traditional CNN is the softmax layer, which effectively chooses the most likely classification based on the output of the FC layer. Although the comparison of 'which likelihood is larger' is computationally very easy, this layer also involves an exponential function that calculates the classification confidence value.

Computing exponential functions in hardware commonly requires either a very large LUT or using an iterative Taylor expansion [191]. As our only objective for the ECG application case was to perform classification, we deferred this step to an external processor (e.g. the local MCU on the Elastic Node or the cloud). Using the outputs of the FC layer directly, a classification can be accurately chosen by taking the most likely class.

Fixed Point Optimisation Multiple researchers have shown that a CNN can be accurately computed using fixed point operations [6, 137]. However, due to the reduced ability of fixed point to represent very small or very large numbers (when compared to floating point), it is crucial to investigate the bit width requirements of the application being created.

Therefore, we created a semi-automated system that aims to find the optimal representation. This involves firstly finding the largest number that needs to be presented. Technically, each addition requires an extra bit to capture overflow ($\mathbf{Q}a.b + \mathbf{Q}a.b \rightarrow \mathbf{Q}a+1.b$) and the result of every multiplication combines their sizes ($\mathbf{Q}a.b \times \mathbf{Q}c.d \rightarrow \mathbf{Q}a+c.b+d$). However, this leads to massive expansions in representations when performing multiple subsequent mathematical operations.

An overview of our process for finding an appropriate representation is provided in the Evaluation in Section 7.2.1, as well as the impact of different representations on overall accuracy. It consists primarily of a set of simulations that evaluate overflow and errors for a given dataset when using various fixed point representations.

The design of the CNN hardware accelerator serves as a good example of the optimisation process when converting an existing algorithm to be compatible with the Elastic Node. By optimising the structure and implementation of the HDL created, a highly efficient accelerator can be achieved without increasing development complexity when a

different neural architecture is to be deployed. In fact, the solution we presented [35] includes a tool for directly converting an existing TensorFlow model. A similar approach was taken in the LUTnet project [70].

5.3.3. IPCA

Our final use case considers the eigenproblem as it pertains to computing eigenvectors and -values from matrices. Although popular over many years, it has also seen a recent boost in usage within applications ranging from categorisation in data analysis [12, 147] and fault diagnosis [281] to image processing. It works by reducing the dimensionality of the data through finding a new set of variables that contain the same information as the original [225]. One of the most common techniques used for this is PCA [189], which captures the variance of a set of variables.

Our primary challenge for solving the eigenproblem on an embedded platform using a hardware accelerator was its high computationally complexity, and the resulting resource requirements. Therefore, considerable effort was required for reducing the resource footprint when compared to other hardware accelerator implementations from the literature [89, 121].

Technical Overview The primary objective when using PCA is usually data size reduction. It aims to reduce the dimensionality of a dataset without reducing the information contained within it. A simple example would be a cloud of 3D points that could be represented adequately when projected onto a 2D plane.

By using the covariance matrix $C = A^T A$ of size $n \times n$ instead of the original dataset A, the dimensionality of the problem is considerably decreased. By retrieving a set of n orthogonal eigenvectors $\vec{u}_i$ that describe the full variance of C, along with their respective scalar weights (or eigenvalues) λ_i, the so-called eigenequation

$$C\vec{u}_i = \lambda_i \vec{u}_i \tag{5.9}$$

can be created. This demonstrates how each eigenvector applied to the matrix applies a linear transformation, effectively scaling it by the eigenvalue. The benefit of this is that a smaller subset of the eigenvalues and eigenvectors can be chosen to cover *most* of the variance in the original dataset [241] – effectively balancing how accurately a newly projected dataset needs to capture the variance of the original.

Traditional PCA's batched nature and computational complexity generally makes it ill-suited to implementation on an embedded device. IPCA [11] improved on this by only computing the incremental impact of incoming data instead of recomputing the full PCA. By projecting incoming data onto the reduced eigenspace, online learning is made possible without growing the original dataset. The difference between the two approaches is shown in Figure 5.3, showing the subtle but critical difference when new data is introduced.

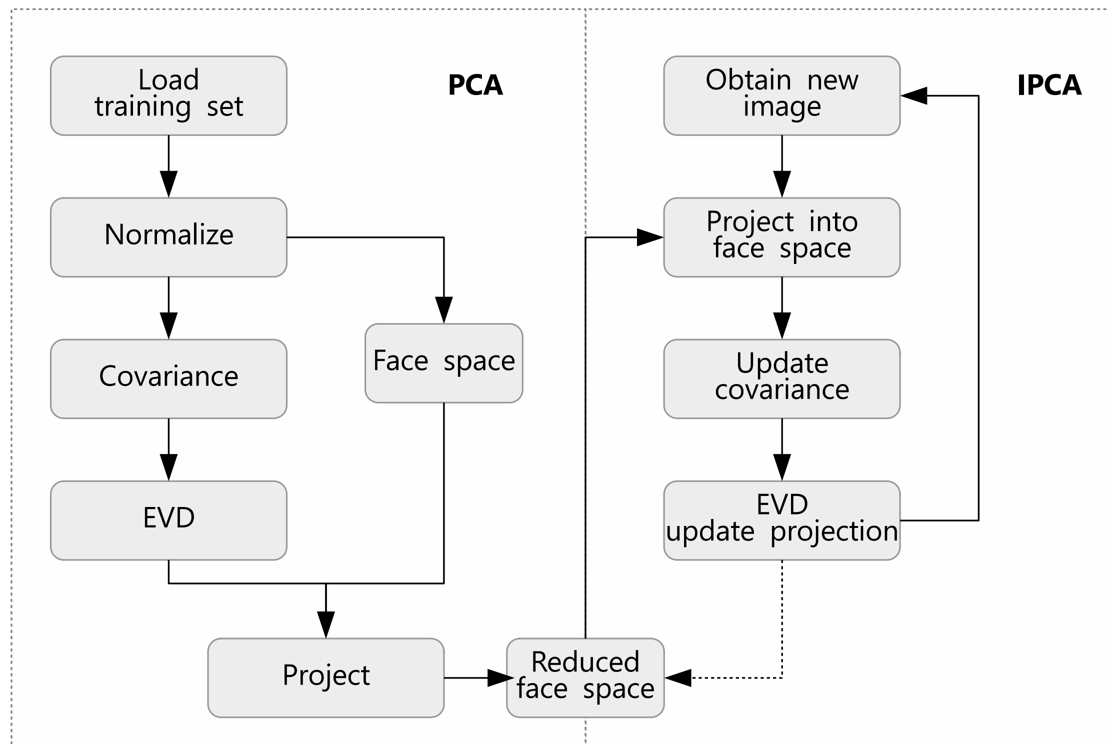


Figure 5.3.: Comparison of PCA and IPCA computation processes

We identified a use case for this in on-device computer vision on drones (more specifically UAVs). Since they are dependent on battery power for both their onboard processing and powering their motors, energy efficiency is at the forefront of their design objectives [26]. However, they have high computational requirements as they commonly operate outside convenient communication range and therefore rely at least partially on local processing or preprocessing. This intelligence enables them to operate autonomously, without depending on a constant connection to a server.

Therefore, we developed an IPCA hardware accelerator [36] that is capable of processing up to 16x16 covariance matrices – allowing the computation of the 16 most important eigenvectors. Depending on the application use case, we found this to be an adequate level of detail for performing basic facial detection. The advantage of using eigenvectors for this application is that it simplifies this highly complex problem to the comparison of the relative weights of the computed eigenvectors.

Our objective with the IPCA is to fit a complex AI component into the resource limitations of an embedded FPGA which was previously not possible. Our approach to this was a combination of well-known hardware accelerator design optimisations and using more modern techniques like QR decomposition and Squared Givens Rotation (SGR) in a novel way.

SGR and QR decomposition As IPCA relies strongly on hardware-expensive trigonometric equations, most FPGA-based eigenproblem accelerators from literature depend on the COrdinate Rotation DIgital Computer (CORDIC) algorithm [249, 250]. Although offering an optimised alternative to traditional solutions, the CORDIC algorithm still causes high resource consumption and computation slowdowns when implemented on an embedded FPGA [195].

Instead, we aimed to use the very efficient SGR [66] algorithm for performing the matrix rotations required to perform a QR decomposition [79]. This has previously not been possible on an FPGA due to the scaling issues introduced by each iteration of the SGR algorithm. By carefully studying how each value needs to be scaled back, we created a considerably more resource and power efficient design capable of being deployed to embedded FPGAs [36].

Our overall design is provided in Figure 5.4, showing how the SGR output needs to be scaled after every iteration to be fed back into the QR Algorithm. It also shows the QR array that performs the matrix rotation that forms the core of our solution.

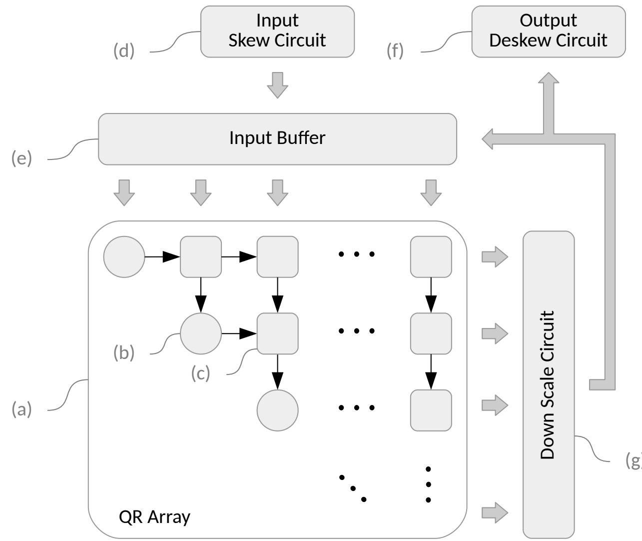


Figure 5.4.: System Overview of IPCA HWF

Each iteration in the QR algorithm on a real symmetric matrix A_0 of dimensions $n \times n$ can be given with

$$[Q_i, R_i] = qrd(A_i) \tag{5.10}$$

$$A_{i+1} = R_i Q_i \tag{5.11}$$

with A_i being each iteration on the rotated matrix. The orthogonal matrices Q_i can be used to retrieve the set of eigenvectors Q through

$$Q = \prod_{i=0}^{n} Q_i \tag{5.12}$$

where $Q_0 = I$.

The result of all of this is to rotate the initial matrix A_0 in such a way that after k iterations we are left with

$$A_k = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \qquad Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \vdots & \ddots & \vdots & \vdots \\ q_{n1} & q_{n2} & \cdots & q_{nn} \end{bmatrix} \tag{5.13}$$

which provides the eigenvalues on the diagonal of A_k, and the eigenvectors on the rows of Q. This fully solves the eigenvalue problem for a provided matrix A_0. Since we use the covariance matrix of the input data, the requirement from SGR of A being a real symmetric matrix is accounted for.

This iterative technique efficiently uses the same QR array to compute a variety of different results. As shown in the thesis of Urban [241], loading a matrix into the processing array is equivalent to multiplying it with Q_i. Apart from using Equation (5.12) to incrementally compute the eigenvectors, each individual Q_i can be retrieved by feeding in the identity matrix I.

Similarly, the second part of Equation (5.11) is computed by feeding in the rotational matrix R_i. This creates a highly optimised solution that reuses the same hardware architecture for computing three different results. This is shown in Figure 5.5, illustrating how all three input matrices can be fed into the array.

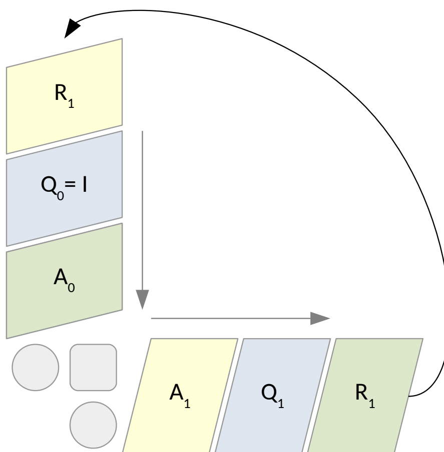


Figure 5.5.: Input sequence $S = A, Q, R$ [Adapted from [241]]

This is a fully pipelined process, further increasing the resource efficiency and computational performance. Although the surrounding logic may increase in complexity, being

able to reuse the same systolic array for multiple different purposes greatly improves resource efficiency.

DSP Timing Optimisations A popular optimisation technique from the hardware architecture design field is to improve the timing of mathematical operations performed using DSP elements. This can have the effects of improving their timing as well as their resource consumption, both of which were utilised for our IPCA HWF [36].

Based on our anecdotal experience when designing this HWF, the maximum frequency of our overall design was increased from 247.64 MHz to 373.13 MHz. By incrementally improving the critical path, the maximum performance possible with the design on that hardware is increased by 50.67%.

Minimising Expensive Operations: Division When developing the IPCA hardware accelerator, we needed to downscale the outputs of each iteration of the SGR. This division operation was essential, and is a primary reason why SGR has not been used by the literature. Since SGR solves

$$\overset{\text{A}}{\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}} \xrightarrow[\text{QR}]{\text{SGR}} \overset{R^*}{\begin{bmatrix} \boldsymbol{\lambda_1^2} & \lambda_1^2 r_{12} & \cdots & \lambda_1^2\, r_{1n} \\ 0 & \boldsymbol{\lambda_2^2} & \cdots & \lambda_2^2\, r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \boldsymbol{\lambda_n^2} \end{bmatrix}} \overset{Q^*}{\begin{bmatrix} \lambda_1^2\, q_{11} & \cdots & \lambda_n^2\, q_{1n} \\ \lambda_1^2\, q_{21} & \cdots & \lambda_n^2\, q_{2n} \\ \vdots & \ddots & \vdots \\ \lambda_1^2\, q_{n1} & \cdots & \lambda_n^2\, q_{nn} \end{bmatrix}} \tag{5.14}$$

instead of the required $[Q, R] = qrd(A)$ as described in Equation (5.11). However, we need to retrieve the true Q and R matrices defined in Equation (5.13).

Inspecting the various elements of R^* and Q^* allowed us to identify the required scaling factors of Q^* and R^*. Each row i of R^* and the column i of Q^* in turn are scaled with ${\lambda_i}^2$, which can be retrieved from the diagonals of R^*.

Calculating the reciprocal square root ($y = \frac{1}{\sqrt{x_{in}}}$) using a very well-known Newton method [140]

$$y_{i+1} = \frac{1}{2}(3y_i - y_i^3 x_{in}) \qquad y_0 = 0.5 \tag{5.15}$$

allows us to avoid explicit divisions. Relatively speaking, these operations are considerably more resource efficient to implement in hardware than divisions.

Another example of avoiding division operations from our IPCA hardware accelerator is inside the systolic array shown in Figure 5.4. SGR requires two divisions [241] for specific internal nodes, leading to a total number of $\frac{1}{2}(n^2 - n)$ dividers for a matrix size of n. Implementing all of these units would lead to a very large resource consumption, since we found that for 32-bit fixed point each divider consumes 1583 logic cells [241].

To alleviate this cost, we investigated the temporal usage of these dividers as shown in Figure 5.6. Note that SGR processing elements only require division in the diagonal

mode [241]. This allowed us to limit the number to only one per row, which reduces the required number of dividers for a matrix size of 16×16 from 120 to 16.

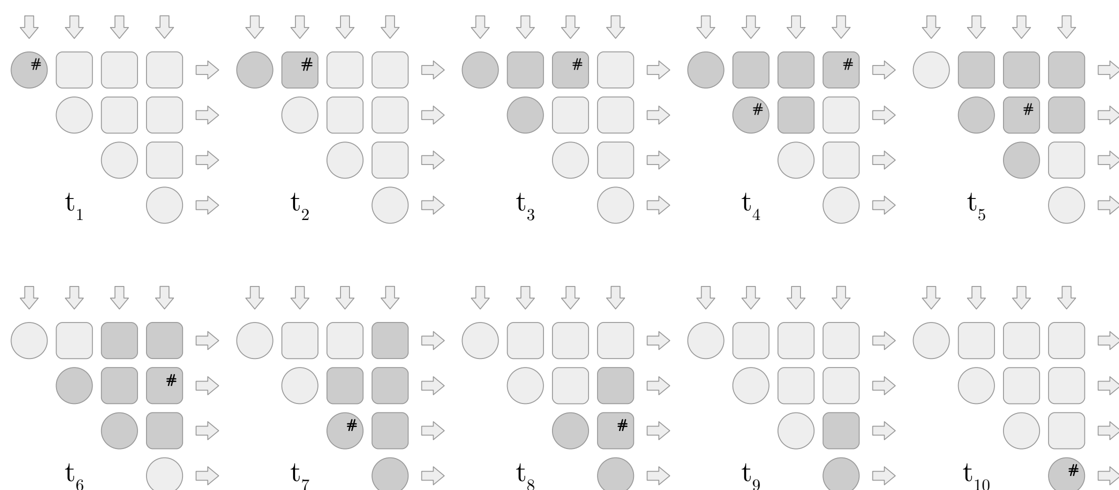


Figure 5.6.: Timing analysis of dividers in SGR systolic array, showing that only one division is required per row

When complex computation operations such as divisions cannot be entirely avoided, the best option is to limit the number that needs to be instantiated. As shown, this can be done by sharing these circuits among different parts of the architecture.

Utilising LUTs for Precomputation The Newton method showed in Equation (5.15) is a very resource expensive operation. Used in the IPCA hardware accelerator, they rely strongly on the number of iterations required to reach a reliable result. Therefore, we optimised by varying the initial guess based on the input value. By taking inspiration from the programming of the well-known classic video game Doom[1], we created a LUT of initial guesses. The speed of convergence is then scaled relative to the number of elements of this LUT, allowing us to effectively reduce this iterative solution to a direct computation (given an adequately large number of initial guesses).

Upon analysis of this we created Figure 5.7, showing the estimated value of $\frac{1}{\sqrt{x}}$ with our estimated "$InvSqrt(x)$". For this example, it is clear that the result is very accurate. The error peaks from time to time, but the absolute error never exceeds 0.038 for the estimated reciprocal square root. Note that this is not the inaccuracy of the initial guess estimation, but instead of the computed Newton method after two iterations of Equation 5.15.

[1] `https://github.com/id-Software/DOOM` (last visited: 2021-12-06)

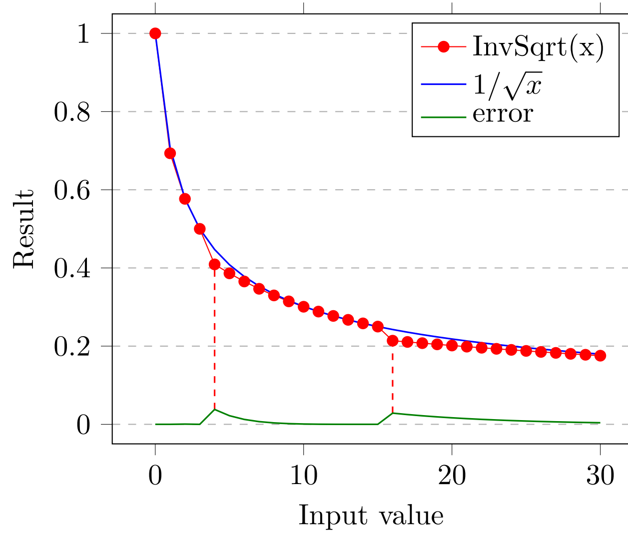


Figure 5.7.: Error analysis of reciprocal square root LUT, showing that the errors are less than 0.05 across all input values.

5.3.3.1. Training and Querying Latency

Each step of the SGR algorithm requires a number of steps, each of which introduces its own latency. The total latency can be written as the sum of the latencies of each of these steps:

$$L(n,k) = L_{FIFO} + k \times (L_{QR} + L_{Sqrt}) \tag{5.16}$$
$$= 24nk + 3n + 6k - 1 \tag{5.17}$$

where each step's latency can be seen in Figure 5.8. This then creates a latency parametrised by the matrix width n and the number of iterations until convergence k.

The last step is then to simply convert this to a throughput value for simpler comparison with related work [89]. This requires the maximum clock frequency to calculate the maximum throughput T

$$T(n,k) = \frac{f_{max}}{L(n,k)} = \frac{f_{max}}{24nk + 3n + 6k - 1} \tag{5.18}$$

which provides the number of full solutions of all eigenvalues and -vectors possible per second.

Our highly optimised IPCA HWF shows what is possible in embedded FPGAs when using appropriate accelerator designs. Along with some careful tweaking of implemented mathematical operations, our novel SGR-based design creates a very fast design that can

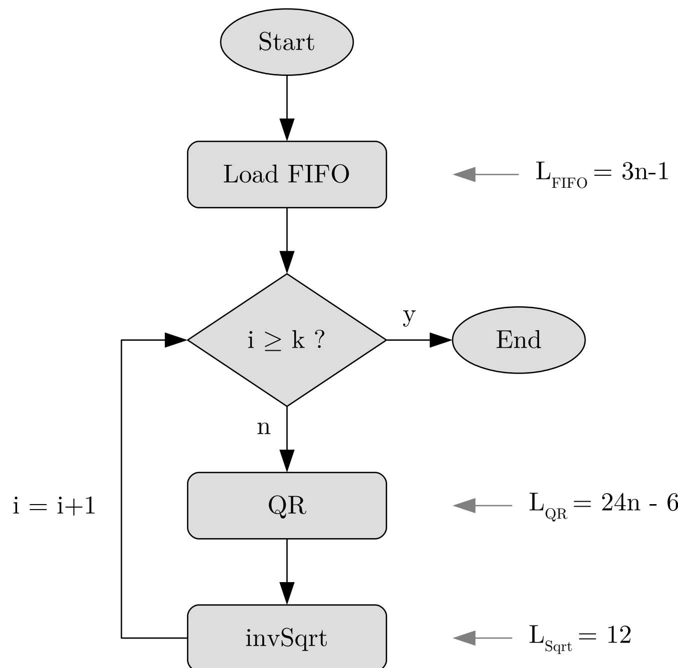


Figure 5.8.: Flow diagram of IPCA process with latency of each step

scale from very small embedded FPGAs to larger variants. The resulting performance and resource utilisation will be evaluated in Chapter 7, along with an investigation on how effectively it can solve the facial detection problem in UAVs.

5.3.4. Summary

We have shown here through these three examples how different hardware accelerators can be effectively and efficiently implemented on the Elastic Node platform. They serve to demonstrate three different approaches that can be followed by the designer, varying primarily in the amount of manual effort involved:

1. ANN — minimally tweaking the design of an existing (larger) design to fit the resource constraints of smaller embedded FPGAs,
2. CNN — parametrising the design so it can be repurposed to a wide variety of devices, and
3. IPCA — designing the entire solution from the ground up to use efficient design principles (e.g. systolic arrays).

Using different combinations of the optimisations covered in Section 5.2, we have adapted each of these examples to utilise all the available resources on the Elastic Node

FPGA. Through this, we also showed that highly complex and capable hardware accelerators can be incorporated – offering great increases to local intelligence over basic MCU-based devices. This allows the user to utilise sophisticated ML and AI in their distributed applications, but so far only on a single device. Effectively utilising teams or groups of devices further requires self-optimisation and other self-x techniques, implying an even higher level of intelligence.

Chapter 6.

Learning Intelligent Devices

The development of the Elastic Node runtime in Chapter 4 provided us with a heterogeneous FPGA-augmented IoT device to use in our experiments. While this was focussed largely on the hardware platform and the software development support provided by the Elastic Node middleware, Chapter 5 emphasised the hardware accelerators to be deployed to its local FPGA. Together, these things allowed us to put highly complex applications locally on our embedded device – improving local *application intelligence* through machine learning techniques such as neural networks and computer vision.

However, local computation on the device is not always the best option. In some cases, the energy overhead of keeping the FPGA constantly powered or reconfiguring it can reduce overall efficiency. Therefore, in this chapter we introduce some alternative behaviours to the Elastic Node, and an optimisation scheme for the device to learn how to choose between these options. This is based on the *offloading problem*, which considers the decision between sending a specific computational task (see the System Model in Section 2.2 for the definition of a task) to another device or computing it locally. A further discussion on the offloading problem was presented in Section 3.4.1.

Consider for example having multiple FPGA-augmented IoT devices available in the same area. It may be beneficial for them to cooperate by *offloading* work to one another. By working together, they could more fairly share the workload to balance their energy usage, and to take better advantage of optimisations such as batching (see Section 3.2). A similar argument could be made for devices offloading to the edge/cloud when no devices are available locally to perform a workload.

This increases the complexity of the decision-making of the devices when compared to local-only solutions. Here we introduce our device *agent* that is responsible for controlling this high-level behaviour of the device: how to handle incoming workloads, controlling the energy states of the device etc. This agent should increase the *device intelligence* by optimising its behaviour to achieve some developer-provided *goal*. As stated in SREQ_{II}: Automated Cooperation Optimisation, its goal is to learn how to optimise the augmented offloading problem so the device can behave intelligently.

Existing solutions for device agents and decision-making (see Section 8.2) do not take into account the presence of local hardware acceleration and the impact that has on optimal behaviour. They generally do not consider P2P offloading – instead focussing on a centralised server-client relationship which may not scale with larger applications

and systems. The agents we developed [34] offer a novel approach to optimising the behaviour of an FPGA-augmented smart IoT device.

Once we formalise the problem we are trying to solve in Section 6.1 and provide an overview of our design choices in Section 6.2, we will set out a list of requirements for our agent in Section 6.3. Next, an analytical model of the system is created in Section 6.4 to facilitate the development of the agent itself in Section 6.5.

6.1. Problem Statement

Consider the use case presented in Section 2.1.2 where a user named Fred owns a number of smart IoT devices in his smart home. These devices are cooperating to perform a number of applications for him – e.g. a neural network-based depression detection that monitors his voice [253]. This process may require a number of steps: detecting and recording the audio samples, pre-processing for certain markers to the presence of a voice [161, 273], and finally a neural network. Since Fred is concerned with the privacy of the conversations in his home, he does not want recordings of his voice to be processed in the cloud and therefore demands that all processing is done within his home network. Sometimes his laptop is available for processing some of the workload, but he uses that for work and therefore does not want it to do the processing.

One solution to this is to have his smart IoT devices share the workload amongst them and divide the application into computational tasks such as recording, pre-processing, and the neural network. Some of these tasks are clearly more computationally intense than others, making the optimisation of sharing the computations non-trivial. Additionally, since the application is dependent on the user and his behavioural habits (such as where in the home he is) the number of *jobs* created can vary unpredictably. In this example, a job would begin with the recording of a sample, and finish once it has been fully processed.

Due to how dynamic both the device's environment and the application itself can be, the agent needs to be *self-adaptive* [33] so it can alter its behaviour based on these changes. Therefore, it needs to support runtime learning so it can create or update its decision-making policy after deployment. This is in contrast to design time learning which is done entirely before deployment.

Another important concept here is the *goal* of the agent's optimisation. Since this goal is dependent on the user's demands and wishes, it would be infeasible to design the agent to be able to address any goal. Some examples of possible objectives could be to ensure the device survives on a single battery charge as long as possible, minimising the latency involved in processing a job, or to process as many samples as possible within a given time frame.

Similar to the goal-driven agents introduced by Russell and Norvig [206], our agents aim to choose an intelligent action based on the current state of the system. These actions should include everything involved in optimising the behaviour of our Elastic Node, including when to deploy a hardware accelerator, which one to deploy when an

application requires multiple different types of processing, and how to handle an incoming job. The overall objective of this is to have the agent act *rationally*, which is defined by Russell and Norvig as the ability to perform the correct sequence of actions that impact the environment in a beneficial way. In our case this environment can also include the device's own state such as battery life, usage level, and hardware state.

6.2. Design Rationale

The domain of intelligent devices offers a large number of techniques for device AI. Within ML, this can further be broken up into categories. We discuss here some of the options we have chosen (primarily using the definitions provided by Russell and Norvig [206] and Sutton and Barto [228]), and why those design decisions were made. The design was originally published at the ACSOS 2020 conference [34], and expanded to a journal article in the ACM Transactions on Autonomous and Adaptive Systems [37]. They were also featured in the context of self-integrating and self-adaptive systems in the Future Generation Computer Systems journal [33], where their ability to adapt to changes in the environment provide additional flexibility.

We specifically focus on unsupervised learning (see Section 3.5.4), since it allows the device to either learn entirely or partially after deployment – simply by being aware of its own state and that of its surroundings. In the literature a number of different approaches have been used to learn optimised device behaviour – particularly with relation to the offloading problem. The evolution of these approaches will be discussed in Section 8.2, culminating in the current state of the art primarily utilising RL.

Q-learning was chosen for this project due to its popularity in the community and since it does not require a full history of states and actions, or a full model of the system or a probabilistic model of the transitions (see Section 3.5.4.1). It has the *Markov property* (future states only depend on the current state), meaning that we do not need to store a full state history of each experiment. This makes it much more realistic to deploy onto an embedded device when it should perform *online learning*.

Another design decision is choosing *decentralised* learning due to the scale of our application cases. When considering larger ones such as smart cities, system scaling becomes a major concern. When using purely centralised learning, all required information (e.g. system state) needs to be transmitted to a single location, and the results need to be propagated back to all the devices. Especially in larger systems – or ones that are more physically spread out – this can cause bottlenecks in either communication (due to interference) or processing (due to limited computational resources).

Lastly, where the agent is deployed should be considered – commonly either in the cloud or locally on each device. This choice is linked to the solution's computational requirements, as well as the communication overhead. Especially when using very resource-limited devices (or an agent requiring excessive data such as a full system state of all devices), a local deployment may be impractical. However, we much prefer a device-local solution, since our system aims to offer local AI as stipulated in SREQ_I.

6.3. Requirements

Somewhat separate from the requirements for the overall system laid out in Section 2.4, a number of features and functionality are required from the agent governing its behaviour. These sit alongside the existing System and Platform Requirements, augmenting them by elaborating on the ones that focus on the agent – $SREQ_{I}$: Local Intelligence and $SREQ_{II}$: Automated Cooperation Optimisation.

It is important to note that the objective of this work is not to create a single agent that always acts in the perfect way. This would be infeasible as applications vary wildly, and what behaviour is desirable depends on the objectives of the designer. Therefore, our primary objective is to provide the developer with the tools required to design their own agent that targets their specific *intent*.

$IREQ_{I}$: Rational behaviour

The agent should be able to govern the device(s) in a rational way. In some cases this is related to optimality and optimising some measurable performance metric, but it extends to behaviour that appears intelligent or 'correct'. Conceptually, a developer should be able to direct its behaviour in such a way that the agent's behaviour 'makes sense'. Although difficult to evaluate quantitatively, this ties in directly with our research hypothesis in Section 1.1 by introducing intelligence to the device's behaviour.

$IREQ_{II}$: Generalisability to Various Applications

Central to our design is the need for generalisability, ensuring that the system can be reused for a variety of different use cases and applications. This primarily means that the core concepts should be clearly enough defined that it is possible for users to pick it up and adapt to their own applications and systems of devices. Crucial to this is providing a good explanation of all the design decisions made, so a user of the system might know how to adapt it to their own needs.

$IREQ_{III}$: Realistically Implementable

Although much of the design process will take place in simulation, this work is specifically targeted at creating realistic devices. That means that all of its functionality should be implementable in a real deployment. Specifically, this means that the agent can only act on information it would realistically have access to. This is of particular importance in decentralised applications where no single device has a complete overview of the entire system. Although many techniques exist for collecting this type of state (e.g. flooding [229]), these would not scale.

IREQ_{IV}: Dealing with catastrophic failure

An agent must be prepared for any of their neighbours to suddenly fail and disappear from the network. Apart from the networking implications of not being able to reach another device, agents should be able to continue efficient operation even when some or many of its neighbours are unavailable. That is a constant danger for the battery-operated devices that our system is designed for, as they have a finite store of energy that can run out an any point. This requires the agents to perform online learning, constantly reevaluating their learned behaviour and updating it as required.

Although not a comprehensive set of requirements, these highlight the most important parts of our design rationale. Together they describe the minimum set of features that an agent should possess for us to consider their behaviour intelligent.

6.4. Analytical Model

The design paradigm of our agent is based on a classic cycle of design, simulation, evaluation, and deployment as shown in Figure 6.1. The distributed system developer creating a new agent starts with an initial design, followed by a simulation of its behaviour within a set of representative scenarios. Based on the results of this simulation the process can either continue onto a deployment that verifies desired behaviour, or cycle back to another iterative design cycle.

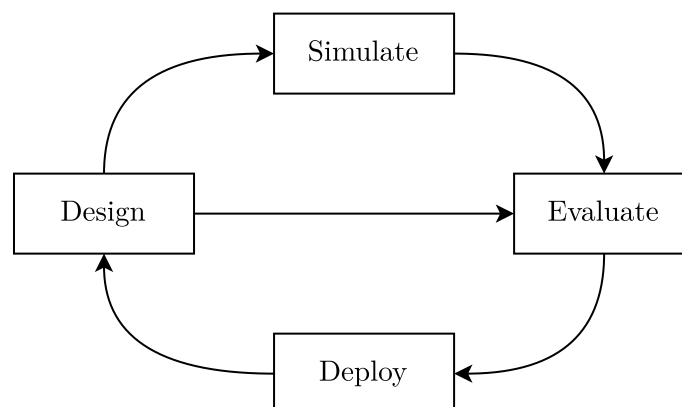


Figure 6.1.: Design cycle describing the iterative process of designing, testing and evaluating a system. Once it has been evaluated thoroughly, its design can either be updated or it can be deployed.

This highlights the importance of an accurate simulation. It relies on an accurate analytical model that represents the environment, the performance and behaviour of the device (or set of devices), and the interactions between them. We use this model only

during the initial simulation phase to develop our agent's policies, some results of which can be seen in the evaluations presented in Section 7.3.

This is fundamentally different to a greedy approach that might use the simulation model on the device for decision-making. It might instead directly compute the costs of each possible action on the device, while we rely on the self-awareness of the device [133] to consider the system state. This reduces the reliance on having a perfectly accuracy model, instead relying on machine learning to adapt to any unforeseen environmental or device internal impacts. Note that the agent's environment and system state can incorporate both internal and external factors [228], essentially providing the developer with the freedom to include any information that may be useful.

Formally, let us consider a set of devices $\mathbb{D} = 1, \ldots, D$ where each device d incorporates at least a local FPGA, an MCU and a wireless transceiver (to allow communication between devices and with a cloud/edge server). Any one of these devices d_i can complete a hardware accelerated job j that is one of a set of possible jobs $\mathbb{J} = 1, \ldots, J$. This is dependent on its FPGA being in the correct configuration at that point in time, as every job j includes a corresponding computational task T_n that requires a specific configuration.

The assumption of having maximum one accelerator available at a time on the local FPGA comes from wanting to use the smallest FPGA possible, and therefore allocating as much of the available FPGA resources to the computational task. This means that the FPGA must be reconfigured each time a different task is to be performed [52] – a cost that is somewhat mitigated if the FPGA is turned off between workloads to save energy. Additionally, each job only includes a single computational task, but this limitation can easily be circumvented by spawning follow-up jobs after a job is finished to represent multi-phase problems.

This leads to a job being defined by the tuple $j = (t, d_i, s_n, T_n)$, where t is the timestamp the job was created, d_i is the device that created it, and s_n is the input data size. Each of these jobs j is then additionally defined as a set of subtasks $\mathbb{S}(j)$ that must be completed in order to complete it. These include both MCU-based subtasks such as data capture and preparation, wireless subtasks such as offloading a job from one device to another, and FPGA-based processing tasks. Some subtasks include a combination of these components (e.g. moving computational results from the FPGA to the MCU). Therefore, each subtask defines the power state $\mathbb{P} = active, idle, sleep$ that each component on the device should be in, where each subsequent state has a lower power usage. For example, the result fetching subtask requires the FPGA and MCU to be active, while the wireless transceiver can sleep to save device energy.

This allows us to formulate the total instantaneous power of a device with N components during a subtask to be

$$P_i^t = \sum_{n=1}^{N} P_n^t \tag{6.1}$$

where P_n^t is the power of component n at timestamp t. For some components this is fairly simple (e.g. the MCU usage is fairly constant under the assumption that frequency scaling is not available), while others such as the FPGA are dependent on the exact computational task being performed. Therefore the component power is expressed in the worst case as a function of the current subtask S and the complexity C of the current job j

$$P_n = f(S, C_J) \tag{6.2}$$

which in some extreme cases can be very non-linear. The FPGA has an extra power state *reconfiguring* which represents the maximum power consumption in the SRAM-based FPGAs we use [216], and occurs when switching to a different configuration.

We assume in Equation (6.1) that the power consumption of the different components is independent, which is not necessarily always the case. Through our testing of the Elastic Node, the power consumption of each component indicated some dependency on that of other components. However, these were small enough to not significantly alter the power distribution, which we approximate with a Gaussian distribution. This is based on the central limit theorem [84], which for a noisy sensor suggests that a Gaussian distribution is a good approximation under continuous statistically independent sampling. Each power state of every component is individually characterised, allowing us to capture variations around the nominal.

Lastly, we require the time consumed by each subtask $\delta(S)$ in order to expand Equation (6.1). We compute the energy consumption of device d_i as

$$E(j_n, d_i) = \int_t P_i(t)dt \tag{6.3}$$

$$= \sum_{S \in \mathbb{S}(j_n)} P_i^t \times \delta(S) \tag{6.4}$$

which considers the energy consumption of a job as the sum of the consumption for all its subtasks. This only leaves the modelling of subtask duration $\delta(S)$, which must be separately done for each subtask. Some examples of this includes the computational time

$$\delta_n = \frac{C_J s_n}{f_{i,FPGA}} \tag{6.5}$$

that uses the task complexity, the input data size, and the frequency of the FPGA. This shows the complexity as a linear scaling value that represents the number of clock cycles required for an input data size of $s_n = 1$.

Here the complexity is assumed constant for different data sizes leading to a linear relationship between δ_n and s_n. This is done for computational simplicity, but non-linearity can be trivially represented by changing the complexity from a constant to a function (e.g. $C_j = f(s_n)$).

The total energy cost of a job $E(j_n, d_i)$ is of critical importance, as this is one of the main optimisation objectives in distributed embedded systems. Under the light assumption that a job does not have a constant energy cost (and therefore that the actions of the device have an impact on this energy cost) this augments the binary offloading problem to optimise simultaneously the number of jobs performed and the total energy cost.

6.5. Agent Design

As stated in IREQ_{II}: Generalisability in Section 6.3, our objective with this design is to make it as reusable and adaptable to different application cases as possible. This requires careful definitions of what is required to create an agent, and for improved convenience it should require as little development effort as possible.

The foundation of our approach lies in the goal-based agents defined by Russell and Norvig [206] as shown in Figure 6.2. Through a combination of inputs from the environment and the state of the world, they choose actions based on a set of action rules. These actions then impact the environment in a certain way (which from the agent's perspective can be unpredictable), and the cycle continues. This design highlights the crucial parts of our design to be the modelling of the world, and creating the action rules themselves.

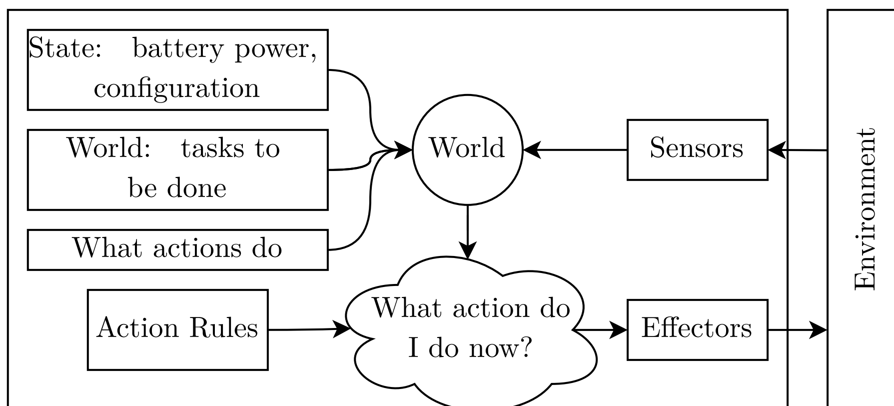


Figure 6.2.: Conceptual overview of agent and environment (adopted from [206]), describing the flow of information and decisions

In our design, a policy π provides the mechanism for the agent to choose an appropriate action. This is an implementation of Q-learning (as discussed in Section 3.5.4.1), where a mapping is created from each possible system state to the value of each available action. This could then be used in a greedy system to simply choose the most beneficial action.

The overview of this is shown in Figure 6.3, demonstrating how the agent is trained. It shows that it creates the current system state by using the *device* internal information,

the current *job* being executed, and the computational *task* attached to that job. The device information could contain both static device specifications (e.g. it's physical or contextual location) as well as more dynamic variables (e.g. battery state or number of reachable neighbours).

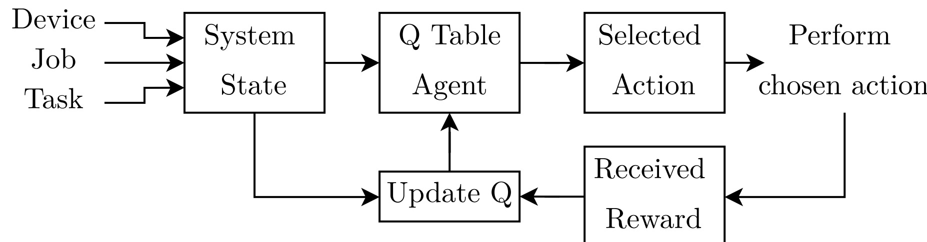


Figure 6.3.: Operational overview of learning agent using a Q-table to choose a new action based on the current system state, while updating its policy based on received rewards.

Composing the system state and action space are critically important steps in the design of a learning agent, and we provide here the current implementation of ours. This is not an exhaustive list of states or actions, and is specifically presented here as an example of how we developed our system. Extending this to a different scenario, set of devices, or even application case would require at least a partial redefinition or expansion.

6.5.1. State-Action Decisions

The first stage in agent design is defining how the agent maps the current system state to a chosen action. This can be done in a number of different ways, leading to different policies being learnt. Each policy can then be used through an algorithm like *epsilon-greedy* selection [228], where exploration (trying a new state to better explore the solution space) and exploitation (using the learned policy to make a rational choice) is balanced through a random probability to explore. Our approach involves two different options for this: either storing explicit mappings in a table called the Q-table, or using a neural network.

6.5.1.1. Q-table

The Q-table is a very simple approach for mapping each possible system state to its opportune action. This requires the system state to be discretised, so that each state refers to one row of the table as shown in Table 6.1. It shows a simple example where two different actions are available, and the system state only has three options. For each state-action combination, a value is stored that represents how beneficial that action would be in the current state.

State	State Index	Action 1	Action 2
State 1	0	0	+1
State 2	1	-1	0
State 3	2	+1	+2

Table 6.1.: Example Q-table with three discrete states and two possible actions, with each state-action combination assigned a Q-value.

For example, when the device is in State 1, the Q-table suggests that Action 2 would be more beneficial than Action 1 since its expected value is higher. As the agent learns, these expected values (or simply Q values) are updated using Equation (3.2) to incorporate any received rewards.

Using the Q-table in this way relies on keeping both the system state and the action space limited, as introducing additional ones can cause exponential growth in the number of elements to explore and store. Pruning offers a mitigation by using generalisation (i.e. considering multiple states or actions to be equivalent) but it still does not overcome the main limitation of Q-tables: What if a new state is encountered?

To address this shortcoming, deep Q-learning [159] offers an alternative where the table is exchanged for a deep neural network. Section 3.5.4.1 describes how this works and how researchers have been utilising it, but our initial objective with this work was to show that even with a basic table one can design an agent capable of optimising device behaviour.

6.5.1.2. Shallow Reinforcement Learning (SRL) Agents

The second option is to learn a neural network to choose an action to create a classification policy (π') which may be different to the one learnt by a Q-table (π). Its goal is to reduce the state-space that needs to be learnt, since an individual value does not have to be approximated for every state-action pair. Instead, it maps the same full system state to the available actions using a neural network. This is beneficial since neural networks are capable of generalising their input when the training data suggests they are synonymous.

Designing the most appropriate network architecture can be part luck, part art and part science. Due to the universal approximation theorem, we already know that it is possible to approximate any non-linear function using a neural network [84]. However, a certain amount of trial and error is still normally required, leading to having to perform multiple iterations. The process normally consists of choosing an architecture, assigning random weights and biases, and testing if it trains to adequate performance. This can then be repeated until an overall performance is achieved.

An engineer has a few tools at their disposal when going through this process, such as reinitialising non-converging networks, creating an ensemble of networks, or expanding the architecture with more hidden layers. Firstly, reinitialising a network can help some

networks that get stuck in local minima and cannot recover. It has been shown that certain initial conditions can stop a neural network from converging [84]. In this case, simply reinitialising their weights and biases with new random values can allow them to find a more opportune solution.

Secondly, an ensemble of multiple networks can be created in order to improve their performance. By averaging out a number of well-performing networks, an even better one can be created [92]. This can be a simple way to find a very high performing solution. Alternatively, the best network can simply be chosen from every group, reducing the impact of bad ones.

Lastly, a deeper network can be created by introducing more hidden layers. Although this increases the number of tunable parameters and therefore the complexity of functions it can approximate, there are a number of concerns with this approach. One problem is that larger networks naturally require more training data to learn properly, which is not available in some cases. A more important issue for us is that this increases their memory and computational footprint, which is a large problem when designing energy-efficient embedded devices. Practically speaking, this creates a hard limit to the size of network that we can use.

Therefore, we focus on using SNNs (as introduced in Section 3.5.4.2) for our novel Shallow Reinforcement Learning (SRL) agent. The network used by our agent is shown in Figure 6.4, highlighting the single hidden layer. This being adequate is partially an effect of using a classification-based approach instead of approximating every Q value. We found that the network would otherwise fixate on unimportant minute differences between losing actions, even though we are only interested in identifying the most appropriate choice.

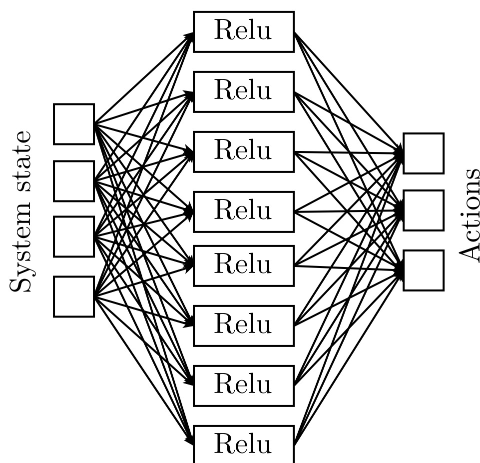


Figure 6.4.: Neural Network architecture for SRL agents. Input vector consists of the current system state as a 1-Dimensional array. The neural network produces a classification output that chooses the opportune action.

To train our SRL agents, we need to "correct" its outputs during training. In a similar way to Q-learning, we use the Bellman equation [228] to update the classification value of the chosen action, incorporating the received reward. As before, this increases or reduces the output value for that action and should over time stabilise to the "correct" action decision. Note that this is not guaranteed to be a truly correct value, but simply a best effort optimisation based on the training data provided. This training data is primarily created using our simulator and analytical model, making it much easier to experiment with different agents when compared to deploying each to a physical device every time.

6.5.2. System State

Defining the system state is a very important step in the design, since it dictates what information is available to the agent when choosing actions. Due to our requirements to have a decentralised and realistically implementable system (see Section 6.3 and specifically IREQ_{III}: Realistic Implementation), this information must be available to the agent when deployed on the embedded device. This may include internal variables such as locally stored data, as well as measurable environmental data.

Each variable is stored in a one-hot binary encoding that is combined to form a 1-dimensional array. This array (as shown in Figure 6.3) is then fed to the Q-learning algorithm to decide the next action. A full example of such a state encoding is shown in Figure 6.5, highlighting how the different states are combined into a single array that describes the current state of the agent and its environment.

Energy Level A fairly universally important state is the current energy level of the device, as it provides context for what behaviour is even possible. When the battery is already critically low, the device should avoid actions that would require a long time (and further energy costs) to pay off. Our system state includes a discrete battery state that can easily be measured by the device (e.g. through the battery monitoring hardware in the Elastic Node platform described in Section 4.5.2.3).

FPGA Configuration As the current configuration loaded onto the FPGA is vital for completing a computational task, we consider whether a specific configuration is readily available. Since reconfiguring to a different accelerator is very similar in energy and time cost to loading a configuration from a cold start, this state is simplified to a logical flag ("is the required configuration for the considered task already available").

Job Queue Length Another important factor in optimising the behaviour of these devices is their ability to efficiently batch jobs as discussed in Section 3.2. Therefore, the agent should consider know how many jobs of a specific type have already been queued. This can either be done using a basic low/medium/high level indicator as shown in Figure 6.5, or a one-hot numeric encoding (one block per possible size).

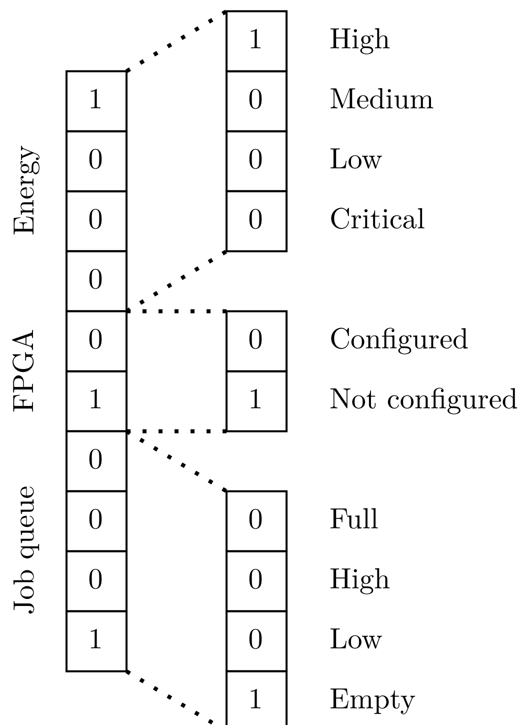


Figure 6.5.: Example encoding of system state, showing a newly started device with a full battery, non-configured FPGA, and no jobs in its queue.

These embody what we consider to be the minimal system state for effectively controlling our heterogeneous devices. Expanding upon this provides the opportunity for the agent to consider different things when making decisions, but also increases the complexity of the decision-making process.

6.5.3. Action Space

As a further attempt to keep the complexity of our agent manageable, the action space is limited to only a few discrete options. This does not aim to be a singularly perfect agent design, but instead attempts to demonstrate that even under these limitations the agent can perform effectively. Our action space is defined as the options that an agent has when a new job arrives (either being created by this device or received from another device).

Batching The first action that can be taken is to simply add it to the local job queue. This is referred to as 'batching', as the primary purpose of this action is to create larger batches for more efficient processing. An example of this action being taken is shown in Figure 6.6, where the incoming job is simply added to a previously empty job queue.

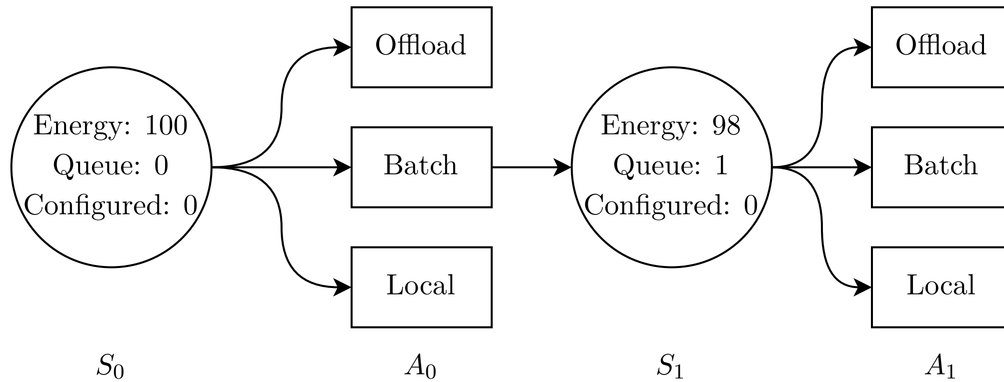


Figure 6.6.: A flow diagram of an agent moving from one state to another, choosing an action based on the current system state. In this example the agent chooses to *batch* an incoming job, which slightly decreases the energy level of the device.

Offloading Alternatively, an agent can choose not to perform a job at all, and instead offload it to a neighbour. Conceptually this might either be a reasonable choice when this device is not well-suited to performing this job (missing hardware or FPGA configurations). Offloading a job then involves sending the metadata describing the job, as well as the data required for processing it (similarly to the Tasklet system [210]). In some applications (such as image processing) this can be a considerable energy expense as a large amount of data needs to be transmitted [26].

Local Processing The final option is to locally start processing the job. This means that the FPGA configuration either already needs to be in place or that it must reconfigure before proceeding. The necessary data must also be offloaded from the MCU (where jobs in the queue are cached) to the FPGA before it can be processed.

Heuristic Behaviour It is important to consider heuristic behaviour, which pertains to actions and choices that are preprogrammed by the user. This may either be as an optimisation step for limiting the number of state-actions (removing specific entries by hard-coding an action-state pair) or as a practical measure that is either required or assumed to be optimal behaviour. For example, we define that once a device starts processing a batch, it will always finish processing its entire queue (consisting of a single type of task). This is based on the energy costs involved in reconfiguring the FPGA to that configuration, and the overhead introduced by reconfiguring it multiple times when this can be avoided [52].

Since our devices have very limited local memory they can only store a finite number of jobs in their queues. Therefore, the maximum number of jobs that can be queued can be fixed at design time, after which point the device is forced to start processing (indicated

by a “Full” job queue in Figure 6.5). Lastly, a special graceful death behaviour is defined to offload all remaining jobs when the battery power reaches a “Critical” energy level. This is to ensure that all essential computational jobs get completed in the case where a device fails.

6.5.4. Reward Functions

One of the most complex and critical parts of agent design is creating the right reward function. This part in particular is generally an iterative process as suggested in Figure 6.1. We present here the reward functions for two different agents that we created, with the aim of (1) illustrating the difference in their performance and learnt behaviours, and (2) to provide insight into the process of imprinting the developer’s intent onto the agents.

These were assembled through information that satisfy three important criteria as prescribed by IREQ_{III}: Realistic Implementation:

1. it needs to realistically be measurable by the device,
2. it needs to be available at runtime, and
3. should either encourage wanted or discourage unwanted behaviour.

Although multiple solutions can be found based on these rules, we set out to create two understandable examples to demonstrate some of the capabilities offered by our system. Our focus on easily available and useful on-device information directly drove our design.

The reward function for the first agent is given by

$$r = R_j + R_e + R_d \tag{6.6}$$

which consists of the job reward R_j, the energy reward R_e and the death reward R_d. The job reward is simply given by

$$R_j = J_{batch} \tag{6.7}$$

and scales with the number of jobs that have been finished (computed and returned to their respective creator) as a part of this batch. The energy reward can be computed with

$$R_e = -\frac{E(J_n, d_i)}{E_i} \tag{6.8}$$

by using the energy from Equation (6.3) scaled with the initial energy of device d_i.

Note that this job energy cost is not computed during deployment using the analytical model described in Section 6.4, but instead is directly measured on the device. Using the internal power monitoring of the Elastic Node platform (see Section 4.5.2) this can

be trivially and cheaply retrieved during normal operation of the device. This is very accurate when compared to most power models, and much more dynamic in unpredictably changing environments. Instead of relying on our small devices to constantly perform expensive and complex energy predictions, they can simply measure the energy costs directly.

Lastly, the reward function in Equation (6.6) includes the death reward given by

$$R_d = \begin{cases} -R_c, & \text{if } d_i \text{ battery critical} \\ 0, & \text{otherwise} \end{cases} \tag{6.9}$$

which uses the nominal critical reward R_c to discourage the device from going into the graceful death state described in Section 6.5.3. This value was chosen experimentally to balance typical reward sizes in R_e and R_j, and represents how important it is for the agent to avoid reaching a low battery state.

This reward function then describes our so-called “basic agent”, which aims to balance performing jobs and avoiding using energy. In contrast to this, our “lazy agent” simply uses the reward function

$$r = R_e + R_d \tag{6.10}$$

meaning that its objectives are simply to avoid using energy and the resulting device failure. This demonstrates how easily the intent of an agent can be altered through the reward function. This allows a developer to experiment with different agents to find one that behaves in the way they want. The ratios of the different elements of this function further provides the user the ability to prioritise one goal over another, providing a further opportunity for fine-tuning.

Agents can be used either in isolation for comparison or together in a heterogeneous environment where they are expected to cooperate. Parallels can be drawn between biological systems such as swarm behaviour and computational scenarios such as a set of drones that must balance system-wide goals (such as exploring an area) with personal goals (such as saving energy).

We have introduced here a design for an agent that can learn how to control the behaviour of the Elastic Node platform. By using reinforcement learning to train either a Q-table or SNN, the agent can target high-level objectives as prescribed by the user. Through a combination of a reward function and a system state, it is capable of not only improving its performance over multiple episodes, but also to respond to sudden change in the environment. Upon such a drastic change, it is capable of adjusting previously learned behaviour to better suit the new environment and situation.

The possibilities of this are tremendous, since meticulously designing rule-based cost-based behaviour is not always feasible. Especially when considering a highly dynamic environment or system, these can also be infeasible or impractical. Instead, being able to define the agent governing a complex device like the Elastic Node using high-level

objectives and goals allows for a greatly simplified development process. Combined with an accurate analytical model and a self-aware device capable of retrospection and self-evaluation, it gives a user the tools required for creating dynamic devices capable of learning rational behaviour.

Chapter 7.

Evaluations

The Elastic Node has been introduced in this work as an FPGA-augmented smart IoT device. It supports heterogeneous embedded applications through its software runtime, and offers a hardware platform ready for experimentation. At this stage we need to address our hypothesis from Section 1.1:

Connected and autonomic FPGA-augmented smart IoT devices can use AI to optimise their behaviour.

This requires us to evaluate each phase of our work – investigating our smart IoT device itself in Phase 1, the FPGA-augmented acceleration it supports in Phase 2, and its optimisation AI in Phase 3.

Additionally, we need to ensure that each of our overall system (Section 2.4), platform (Section 4.2), and learning (Section 6.3) requirements have been met. As a reminder, the overall Functional and Non-functional system requirements that detail what the work as a whole should achieve are

I **(NF) Adequate Local Intelligence to Support Various Applications**: Supporting complex applications without depending exclusively on offloading requires a certain level of local intelligence to compute it.

II **(F) Automated Decentralised Cooperation Optimisation**: The devices should be able to cooperate autonomously with peers, working together to improve their overall behaviour.

III **(NF) High Energy Efficiency**: IoT and other battery-dependent applications demand careful usage of energy to maximise lifespans without compromising performance.

IV **(F) Convenient and Efficient Local Hardware Accelerators**: Changing between FPGA configurations (to either wake it from sleep or support multiple accelerators in the same application) should be both quick and effortless for the developer.

The other sets of requirements will be revisited throughout this chapter as relevant.

Each phase of our project will now be evaluated in turn, starting with the design of the Elastic Node platform itself in Section 7.1. Next, the local intelligence achievable through the deployment of various hardware accelerators is evaluated in Section 7.2. Lastly, the cooperation of various devices is evaluated in Section 7.3.

7.1. Phase 1: Elastic Node Viability

When designing the Elastic Node platform, the overarching system requirement in mind was $SREQ_{III}$: Energy Efficiency. It formed the foundation of the component choice, leading our preference to the smallest and lowest power options available. The challenge then became ensuring that the device performance remained adequate for solving useful tasks. Therefore, the balance between performance and power consumption was absolutely critical.

Section 4.2 specifies the requirements that our Elastic Node platform should satisfy to be viable as an experimentation platform for developing smart IoT devices. These are:

I **Flexible Support for Real World Deployments**: It should create a practical and useful platform for running experiments in the real world.

II **Dynamic In-field Accelerator Reconfiguration and Control**: The platform should provide support for changing and using a variety of different accelerators at runtime, creating a flexible solution for a range of computational problems.

III **Easy and Fast Accelerator Access**: Offloading specific tasks to the accelerator on the local FPGA should be both simple to do and not introduce excessive overhead (both in resource consumption and time).

IV **On-device Energy Measurements**: Smart devices – specifically self-aware ones – require a good understanding of the local system, calling for accurate and detailed live energy monitoring.

V **Easy and Low-overhead Accelerator Reuse**: Both adoption and long-term development convenience relies on making it easy to reuse and adapt existing hardware accelerators.

Each of these requirements will be discussed here, even though some have already been discussed in Chapter 4. For example, $PREQ_{IV}$: On-device Energy Monitoring was demonstrated through the accuracy and convenience of the power monitoring hardware and library in Section 4.5.2. Through a simple API, an embedded software developer can retrieve the power or energy usage for any of the main components on the board.

7.1.1. Energy Consumption

Our requirements for a platform capable of performing realistic experiments ($PREQ_I$: Real World Deployments) and $SREQ_{III}$: Energy Efficiency make it important to consider its energy consumption. In order for it to be useful for mobile or battery-powered experiments, it needs to offer both low *active* and *idle* power consumption. We base our testing on an assumption of a low duty cycle (where the device spends most of its time in idle or using very little power), as is common in various different fields (e.g. Wireless Sensor Networks (WSNs) [46, 88, 204]).

However, an active consumption above the capabilities of small batteries (e.g. small LiIon batteries are designed to be discharged at roughly 164 mA or 600 mW for a typical 3.7 V cell, and some small LiPo ones at $300 - 340$ mA) could cause reliability issues. Batteries commonly are rated both for a maximum instantaneous power draw and a nominal continuous discharge rate (which is commonly much lower).

ANN Power Consumption Firstly, let's use the power monitoring data from Section 4.5.2 to provide context into what class of device the Elastic Node platform is. A breakdown of typical power consumption from different components is shown in Table 7.1, illustrating the detailed multi-component monitoring – as these numbers were all collected directly on the device and then verified using lab equipment such as an oscilloscope. These values are collected from an Elastic Node v3, and illustrate the consumption during a deep sleep, idle, and the active processing of an ANN HWF (as demonstrated at PerCom 2018 [31]).

Component	Sleep Power	Idle Power	Active Power
MCU	3.63 mW	11.78 mW	42.24 mW
FPGA	0 mW	36.93 mW	52.13 mW
Monitoring	2.5 mW	5.94 mW	40 mW
Wireless	3.4 mW	19.34 mW	89.17 mW

Table 7.1.: Typical power consumption for different components

Note that these are all steady state averages, and do not capture the dynamics of short-term power consumption. For example, the FPGA reconfiguration power consumption is known to be higher than the active processing power consumption [244]. This table serves primarily to illustrate that the Elastic Node hardware platform can offer low total device power consumption – in this case around 200 mW under active usage.

In this experiment, each component in the device can be in one of three energy states (sleep, idle or active). *Sleeping* is defined as the lowest power state that can be recovered from without outside influence (e.g. entirely off for the FPGA as it is controlled by the MCU). *Idle* is the resting state that can be instantly woken up from without any delay, while *active* is under full processing load. On the MCU and FPGA this is when computing.

It also shows that the monitoring logic uses less than 6 mW while collecting data (idle) and only increases to 40 mW when it is returning the collected data to a computer (active). Similarly, the 802.15.4 wireless module on the Elastic Node hardware uses less than 20 mW when in idle, increasing to just under 90 mW when actively sending or receiving data. This highlights the importance of minimising the transmission of larger messages such as images or raw audio recordings.

CNN Power Consumption Similarly, we studied the per-component power consumption of the Elastic Node hardware platform when deploying our CNN hardware accelerator [191]. In this case, two different FPGAs from the Spartan 7 family were investigated. Table 7.2 serves to show the differences when using the same accelerator in FPGAs with different amounts of resources – since their consumption is dependent on how their configurable logic is utilised [256].

Table 7.2.: Elastic Node hardware platform power breakdown

Component	Idle	Active S15	Active S25
MCU	2.3 mW	75.4 mW	78.8 mW
FPGA internal	0 mW	16.9 mW	26.6 mW
FPGA aux	0 mW	12.3 mW	18.8 mW
FPGA IO	0 mW	73.4 mW	74.6 mW
Total	2.3 mW	168 mW	198.8 mW

This shows that both the active and idle power consumption of our CNN accelerator fit the limitations of small LiPo batteries. Based on a typical duty cycle of 10%, we could expect a battery life of 16.4 days with an affordable 2000 mAh battery. This would be adequate for experiments, especially if automated charging can be implemented. This clearly indicates that we have addressed our PREQ_I: Real World Deployments, as the Elastic Node can be used to perform realistic experiments on battery power, offering a battery-life of more than two weeks.

Memory Hierarchy An important factor in designing energy-efficient heterogeneous systems is how they utilise the available local memory. Apart from targetting the obvious reduction of overall memory consumed and number of memory interactions, the memory hierarchy can greatly impact the power consumption of a design.

We used a basic FIR filter accelerator to help us experimentally identify the best memory architecture to use for the CNN use case [191]. This includes the decision of what data is stored in volatile/non-volatile memory, and what type of memory technology is used for each (see Section 5.2.2). In a rough order of fastest to slowest, this includes LUTs, BRAM, external SRAM and external DRAM.

Firstly, we investigated the impact of using more BRAM in an accelerator in Figure 7.1. The two impacted power rails were V_{CCINT} and V_{CCBRAM}, which both showed roughly

linear increase as larger buffers were implemented. This shows that relying on more BRAM in your design will inevitably increase the power consumption, highlighting the importance of optimising the amount of fast memory required. For example, the total power increased by 14% when going from 2K to 8K 16-bit buffer size. This is a meaningful difference, considering this power consumption increase will be in effect as long as this FPGA configuration is loaded.

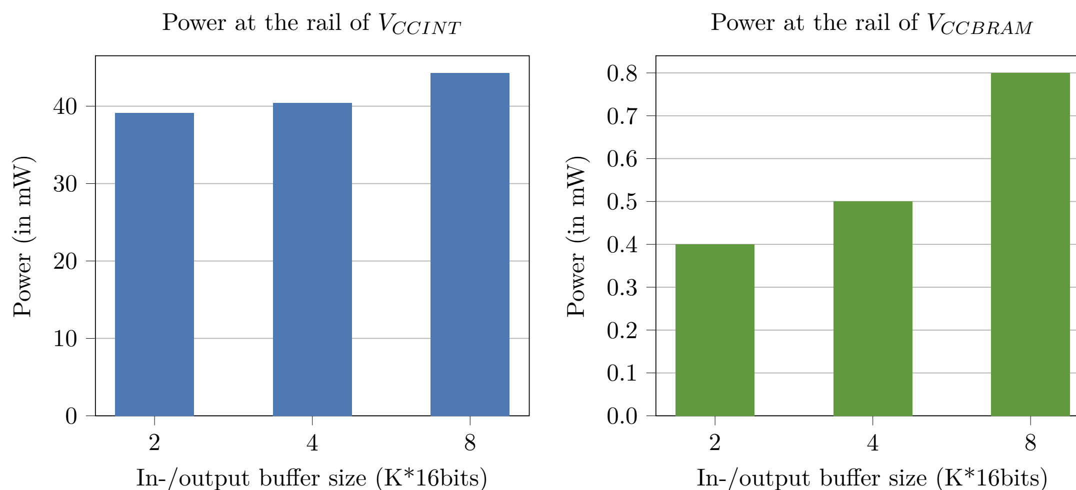


Figure 7.1.: Power consumption at various buffer sizes in the BRAM

Next, another design option was investigated in the structure of the BRAM memory. A number of blocks of BRAM are distributed throughout the device, creating the choice of either fully utilising fewer blocks or distributing it more across the device. Additionally, this would change how the logic is routed on the FPGA, as more logic is used for routing data to where it is needed.

The impact of using different sized blocks of memory to implement a total of 8Kb of memory in BRAM is shown in Figure 7.2. Although less of an impact than using more BRAM overall, using a single larger block of BRAM consumes more power than smaller distributed blocks (up to 60% more power consumed by the BRAM when comparing a single 8K block to four 2K blocks, but only a 7% total power increase). We hypothesise that this is due to the simplified (and shorter) routing possible when using smaller blocks, since the data can be located closer to where it is needed. This likely reduces the overall utilisation of the PL of the FPGA, reducing overall power consumption.

Lastly, we compared the power usage of using volatile memory in the form of BRAM to non-volatile memory such as LUTs. Note that this is not a free choice in many applications – e.g. our ANN accelerator where weights and biases are updated through training and therefore cannot be stored in non-volatile memory as they cannot be easily updated during runtime. However, it is important to study the impact of using the different types of memory so the design can be optimised for speed (as LUTs are considerably faster

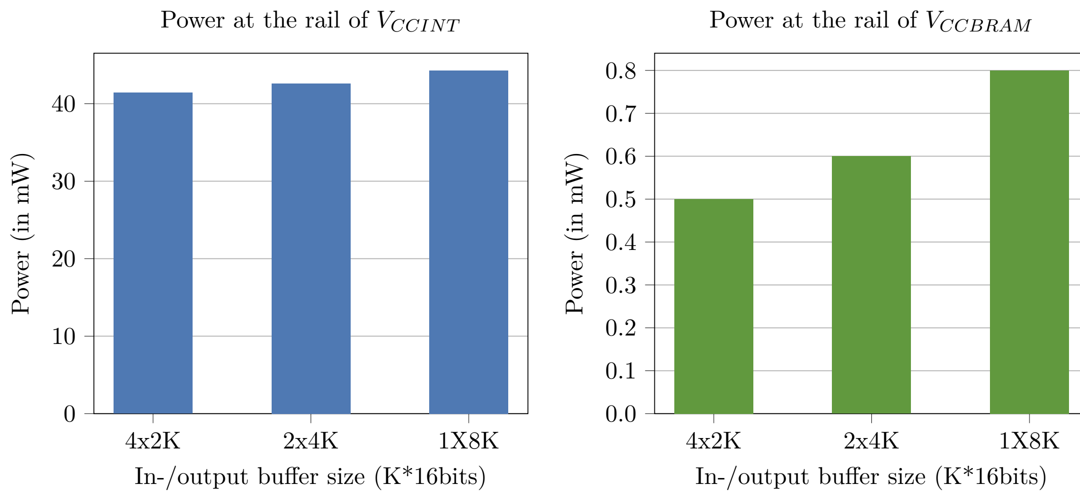


Figure 7.2.: Power consumption when utilising various different structures of BRAM buffer

than BRAM) or power efficiency.

The results are shown in Figure 7.3, providing the total power consumption when using either LUTs or BRAM for various memory sizes. When using more than 1000 words of memory, both options increase roughly linearly with LUTs using roughly 2.3 mW more. This is not a tremendous increase, but with longer active times or larger amounts of memory utilisation the difference could become meaningful (especially if the BRAM has to remain active throughout to hold its data).

Interestingly, for lower memory utilisation the LUT offers a lower power consumption. Only when reaching 500 words of memory does the BRAM win the power comparison. It may appear somewhat strange that the consumption does not change drastically when using 100 to 2000 words of BRAM, but this is simply because the entire chunk fit into a single block of BRAM. Therefore, one may assume that the small discrepancy in usage is dependent not on the memory usage itself but on the routing or controlling logic. When utilising more than 2k 16-bit words, multiple banks of BRAM are required and the usage becomes more linear.

7.1.2. Accelerator Switching Latency

Switching from one HWF to another introduces not only latency in the execution of the application, but also causes energy loss due to overhead. Therefore, it is important to minimise this time and thereby make the system more responsive and adaptive to change. We evaluate PREQ_{III}: Easy and Fast Accelerator Access, SREQ_{IV}: Convenient and Efficient Local Accelerators and PREQ_{II}: In-field Reconfiguration and Control here by studying the activation and configuration switching time of our FPGA. This repre-

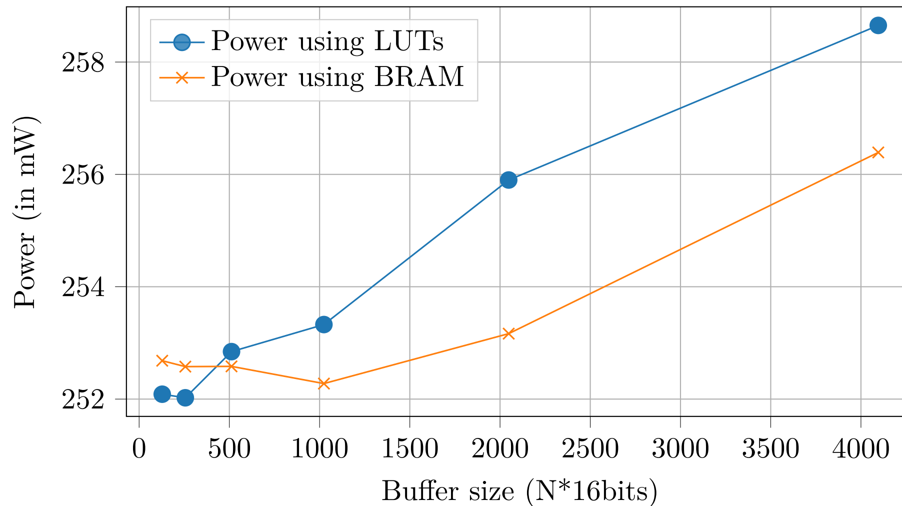


Figure 7.3.: Power consumption for various sized buffers in either BRAM or LUTs, showing that LUTs are more efficient for buffers smaller than 500 while BRAM is more efficient for larger sizes

sents the latency introduced when wanting to offload computations to an unavailable HWF. Note here that the required FPGA configurations are assumed to already be locally available on the device's flash memory, otherwise additional latency would be introduced as the required configurations are downloaded OTA.

The results are presented in Table 7.3, providing the time required for loading an FPGA configuration in various ways on the Elastic Node v3. Firstly, the start-up and online switching times are shown to be identical. These describe the latency when loading the initial image from a cold boot (e.g. when changing out of the sleep state) and using the on-board Internal Configuration Access Port (ICAP) interface provided by Xilinx [262] to switch from one active configuration to another. They introduce the same delay since the time is dictated by the time taken to read the new configuration from the local flash memory by the FPGA – which happens at a known and static clock frequency.

Activity	Latency [ms]
Start-up	94.32
Online switch	94.32
Software Multi-boot	207.45
MCU direct	7150

Table 7.3.: FPGA accelerator switching latency

Software Multi-boot is another alternative, and involves loading an initial generic

configuration in order to load a specific one from an arbitrary location in flash. In this method, the FPGA is started from a cold boot and loads its initial (smaller) configuration file from location 0 in the flash. Then, an online switch is requested from the implemented ICAP module through the Elastic Node middleware, and the FPGA performs a restart to load this second configuration.

Note that this is not the only way to load a specific configuration from a cold start. However, this final method requires flash memory to be shared between the FPGA and the MCU (as is available on the Elastic Node v4), as the MCU alters the initial boot flash location from the default 0 to the location of the requested configuration. This achieves identical latency to 'start-up', but is not available on all platforms as it requires complex SPI multi-master sharing where a single flash storage chip is accessible from both the MCU and the FPGA.

MCU serial is included for completeness, and refers to loading a configuration to the FPGA directly from the MCU through the SelectMAP interface. This involves reading it from flash (or receiving it from a PC via USB), and relaying it to the FPGA. It is considerably slower than the other methods, but could be the only viable option when the flash storage is not directly accessible to the FPGA.

We consider an accelerator switching latency of less than 100 ms to be adequate (especially since it is largely limited by the hardware capabilities of the FPGA and SPI interface). Therefore, the speed requirements of PREQ_{III}: Easy and Fast Accelerator Access as well as SREQ_{IV}: Convenient and Efficient Local Accelerators are satisfied through the latency shown here. Simultaneously, PREQ_{II}: In-field Reconfiguration and Control has clearly been addressed, since the device can switch between numerous hardware accelerators available on local flash storage at runtime.

7.1.3. Elastic Node Middleware Resource Overhead

The resource overhead introduced by our software and abstraction layers should be closely studied. The FPGA resource usage of the middleware and skeleton in particular are of great concern, as this reduces the amount of resources that remain available to the accelerator itself. This is related to SREQ_{I}: Local Intelligence, since excessive resource overhead in the middleware affects the device's ability to deploy complex hardware accelerators. To evaluate this, we created special matrix multiplication HWFs that provide insight into the overhead introduced by various parameters.

By varying the size of the matrices, we can directly change the size of the HWF created, and see how that impacts the FPGA resource overhead introduced by the middleware and skeleton. Firstly, we see the increase in LUT usage in Figure 7.4 that is brought on by the increased complexity of the skeleton. For every additional element in the input matrices, the incoming address needs to be compared with another option in the skeleton.

Similarly, the slice registers that are used for buffering the input for the HWF increase as shown in Figure 7.4. This overhead is even smaller than the LUTs, maxing out at

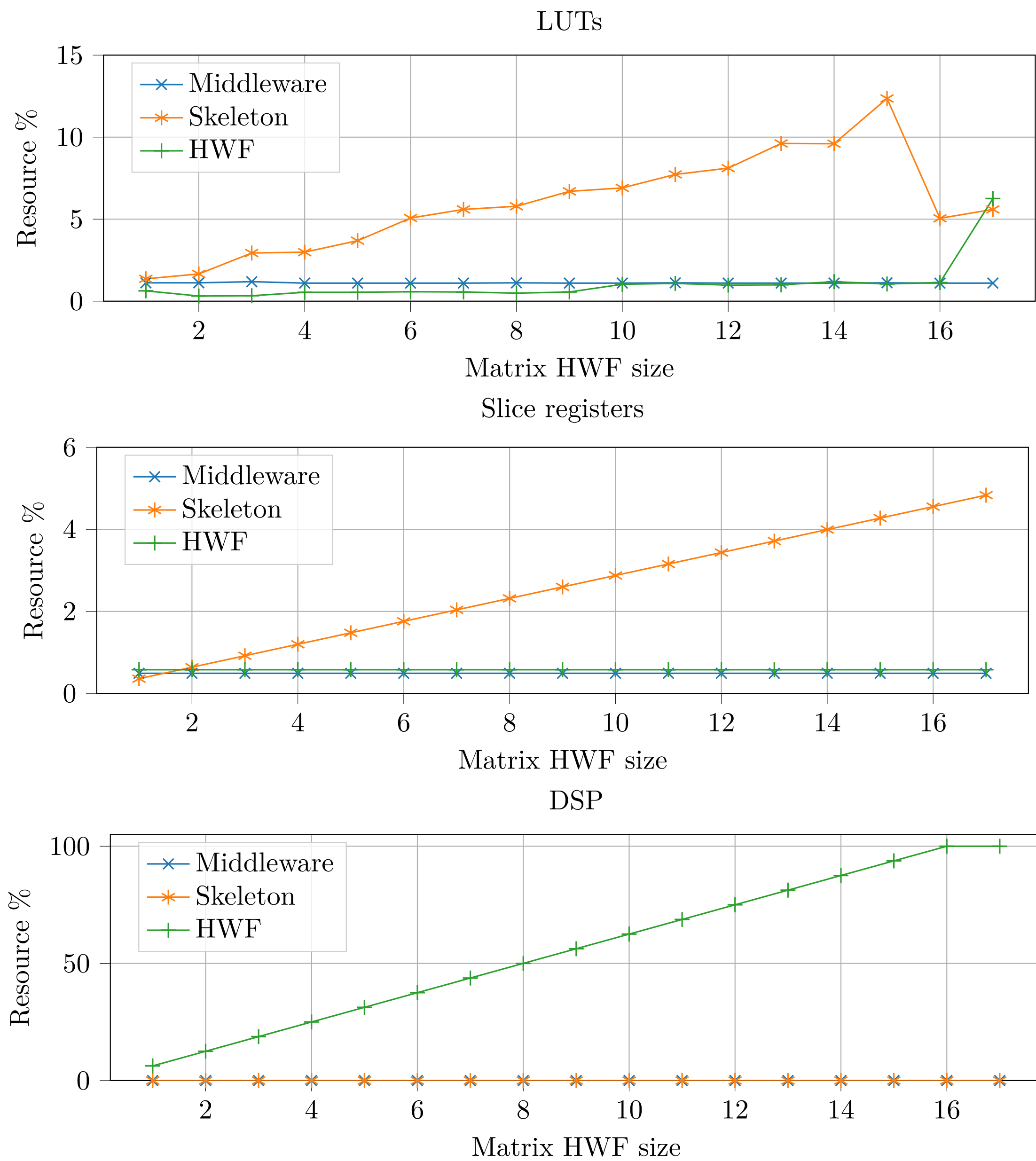


Figure 7.4.: Lookup table, slice register, and DSP usage of different system components, demonstrating that most of the resources are available to the HWF itself

only 5% for an input matrix size of 17.

Lastly, the DSP usage is shown in Figure 7.4 – highlighting that all of the available

resources are provided to the HWF being implemented. It also shows that the usage increases linearly with larger matrices, plateauing after 16. This is reflected in the LUT graph, where the final data point for the HWF spikes.

This causes the strange usage numbers in LUTs and DSP slices for input matrix sizes of 16 and 17. When the synthesizer does not have enough DSP resources available to implement all of the required logic (100% is already used by the HWF at input size 16), it falls back to using the less efficient (and slower) LUTs. This explains why at larger input sizes than 15 (where the complex computations still fit in available DSPs), the LUT usage changes sharply. The decrease in skeleton LUT usage can likely be attributed to those buffering resources being integrated into and combined with the now LUT-based computations.

This shows that the FPGA overhead is minimal, not causing issues for this HWF even at larger input sizes. This is dependent on the specific HWF, as it will require a different skeleton as discussed in Section 4.4.2. Considering these measurements are collected on an Elastic Node v3 with a Spartan LX9, the later versions of our platform would see lower relative overhead as larger HWFs are supported and therefore a lower percentage of their resources should be utilised by our deployed middleware or skeleton.

7.1.4. Development Complexity

In order to assess the convenience part of PREQ_{III}: Easy and Fast Accelerator Access, one must consider the usage pattern of our Elastic Node middleware. We do this here by providing a number of examples of its usage, focussing on the API provided as well as the development overhead involved in ensuring compatibility.

As has been done in a number of our publications [32, 33, 211], we present here examples of the simplified programmatic access using code listings and message sequence diagrams. These include MCU-side stubs and applications written in C, generated FPGA-side skeletons, and IDL definitions.

The objective of our software runtime is to simplify

1. accelerator reuse and access, and
2. management tasks including FPGA control.

Through that, the workload of both hardware accelerator and embedded systems developers should be lightened. By minimising work duplication and development overhead, the reuse of optimised accelerators is encouraged and simplified (as stated in PREQ_{V}: Accelerator Reuse).

7.1.4.1. Using Elastic Node Runtime

As an example, let us consider the final usage API for our ANN HWF. This design was adapted from the larger design created during the FiPS project [50, 75] for the

considerably larger Virtex FPGA [87]. However, the interface of the accelerator as shown in Figure 7.5 remained the same, minimising the system integration effort.

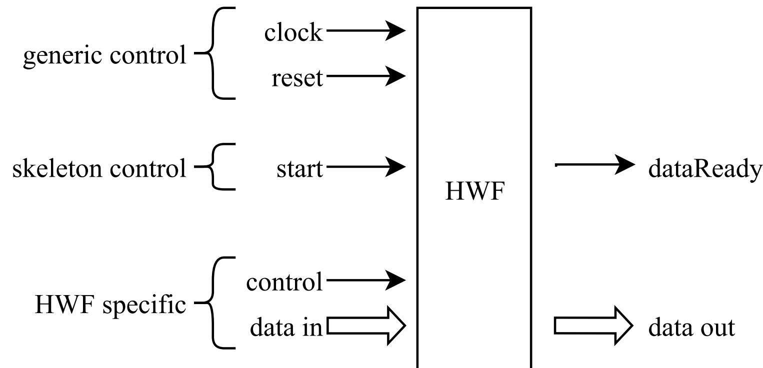


Figure 7.5.: Interface overview of ANN HWF

This shows that the actual interface of our ANN HWF consists of only 7 connections. Two of these are generic control, two are skeleton control, and the remaining three are HWF specific. The `learn` input in this case switches between a inference query and a training run, while the `data in` and `data out` provide the input data to the network and returns the resulting classification in turn. The generic part of the interface (`clock` and `reset`) are present on practically every synchronous HWF, while the `start` and `dataReady` identifies the skeleton type.

The primary development task for integrating a HWF like this into the Elastic Node platform is to create its IDL representation. Although this is currently a manual process, it enables the automated generation of both the stub and skeleton. By using the Extended Backus-Naur Form (EBNF) definition as provided in Appendix A, the API of the HWF can be written down in an intuitive and standardised way.

As a qualitative evaluation of the IDL created, consider the API for the ANN HWF shown in Figure 7.5. It consists of a number of single bit inputs and outputs (e.g. clock, reset, dataReady), and vector-based values such as data in and data out. Each of these need to be mapped in the IDL description to a VHDL and C compatible interface line. This is done in Listing 7.1, which describes how each bit and u8 (8-bit wide vector) is defined in the skeleton. For reference, the full VHDL entity description is provided in Appendix B.1.

Code Listing 7.1.: IDL description for the ANN HWF

```
1 configuration NeuralNetwork_example:
2    mcu:
3       wordsize = 8
4       addresswidth = 16
5       endianness = little
```

```
6       activestate = high

7    function NeuralNetwork:
8       hdl = vhdl
9       endianness = little
10      activestate = high
11      type = oneshot
12      library = work
13      implementation = Behavioral
14
15      bit clk -> clock
16      bit reset -> reset
17      bit start -> start
18      bit learn -> ctrl_in
19      u8 connectionsIn -> data_in
20      u8 wanted -> data_in
21      bit dataRdy -> done
22      u8 connectionsOut -> data_out
```

In this example, an Elastic Node v4 is being used (see Appendix D). The MCU employed is an 8-bit AVR that has a 16 bits external memory address interface. Note that indentation is optional and only aids to increase readability. Based on the discussion in Section 4.4.2 we can identify this as a simple one-shot interaction. This means that the HWF can be interacted with using a simple blocking function call, for example

Code Listing 7.2.: Simplified API calls to the ANN HWF in a generated stub

```
1  uint8_t annTrain(uint8_t wanted)
2  {
3      neuralnetwork_set_learn(1);
4      neuralnetwork_set_wanted(wanted);
5      neuralnetwork_set_start(1);
6      while (!neuralnetwork_query_dataRdy())
7          continue; // wait for hwf
8      return neuralnetwork_query_connectionsOut();
9  }
```

which neatly hides the complex internal control logic. By including the wait for the data ready signal inside the stub call, a synchronous stub is created for the programmer to use. Only the data interface is presented to the user, while the interaction specifics are

defined in the IDL, and impact the internals of the generated stub and skeleton. For further clarity, the full stub and skeleton are shown in Appendix B.

Alternatively, an asynchronous interface could be created using

Code Listing 7.3.: Asynchronous API call to the ANN HWF

```
void (*annCallbackDelegate)(uint8_t);

void annTrain(uint8_t wanted, void (*annCallback)(uint8_t))
{
    neuralnetwork_set_learn(1);
    neuralnetwork_set_wanted(wanted);
    neuralnetwork_set_start(1);

    annCallbackDelegate = annCallback;
}

void annDataRdyInterrupt(void)
{
    (*annCallbackDelegate)(neuralnetwork_query_connectionsOut());
}
```

which exposes the callback to the user. Instead of blocking the stub call until the result is available, the initial stub call stores a callback through a function pointer and returns the execution flow to the MCU application. When the interrupt function is called via an Interrupt Service Routine (ISR) registered to the `data_rdy` pin in the HWF interface, the provided callback is called – allowing the user to receive the data as soon as it becomes available.

Listings 7.2 and 7.3 shows two different ways to interact with the HWF to perform a training run. In each example, only a single run is performed, which is commonly very inefficient (see the discussion on batching in Section 3.2). However, different abstractions can be created that utilise the stub API. For example, when multiple training runs need to be performed in quick succession, the `set_learn` function can be omitted between calls and the output does not have to be retrieved using `query_connectionsOut`.

These examples demonstrates how simple offloading computations as complex as training an ANN can be. By exposing only the header shown in Listing C.2, the complexities are managed by the Elastic Node middleware and remain hidden from the user.

7.1.4.2. Without The Elastic Node Runtime

To provide context, consider the interaction scheme without the abstractions provided by the Elastic Node middleware. As covered in Section 4.3.2, one alternative would be

a message-based system. The abstractions provided are highly dependent on the system used, as it could even provide a similar API to the Elastic Node runtime for creating and decoding the messages required.

A large advantage of using the Elastic Node middleware for offloading tasks is that it uses minimal boiler plate code, and does not rely on a special programming language. For example, the Message-Oriented Middleware (MOM) Tasklet offloading mechanism developed by Schäfer *et al.* [209, 210] uses C-- – a subset of C with additional boiler plate code for defining the behaviour of each task. Although this assists system optimisation, it creates substantial overhead for developers to reimplement their source code in the project's special programming language.

Instead, the Elastic Node runtime aims to provide thin adaptation layers for connecting existing hardware accelerators with the embedded application. This is done through the generated stub-skeleton abstraction, while requiring only the interface description in the provided IDL format. While minimising work redo, this relies on the hardware accelerator developer to optimise their designs for smaller embedded FPGAs.

To demonstrate, consider the functionality being abstracted by the `annTrain` function in Listing 7.2. Most heterogeneous systems rely to some extent on populating a specified area of memory with the needed data: whether that is demarshalling the data in a middleware, populating a message in a MOM, directly interacting through shared memory in a SoC approach, or even setting remote variables in a memory-mapped bus system.

In each of these cases, we would assume that the relevant pointers have been defined. This leads to the manual version (`annTrainManual`) shown in Listing 7.4, which would require the manual definition of each used pointer value. It also still requires the full definition of a skeleton on the FPGA, which can be a large development effort (as can be seen in the generated skeleton in Appendix C).

Code Listing 7.4.: Manual memory interactions version of ANN HWF

```
1  uint8_t annTrainManual(uint8_t wanted)
2  {
3      *NEURALNETWORK_CTRL_IN |= _BV(NEURALNETWORK_LEARN);
4      *NEURALNETWORK_WANTED = wanted;
5      *NEURALNETWORK_CTRL_IN |= _BV(NEURALNETWORK_START);
6      while ((*NEURALNETWORK_CTRL_OUT & _BV(NEURALNETWORK_DATA_RDY)) ==
       ↪   0)
7          continue; // wait for hwf
8      return *NEURALNETWORK_CONNECTIONS_OUT;
9  }
```

Instead of communicating via this memory-mapped interface, one could also use alternatives such as direct pins. Although this would work well for simple binary-based

HWFs (e.g. a binary input ANN), other data-driven ones would be very inconvenient to implement. When sending either memory chunks (e.g. strings or images) or real numbers (either fixed or floating point) this would require a manual interface definition on the HWF to describe how the data is transferred.

While this covers the data interaction for operating the HWF, numerous other features are provided by the Elastic Node runtime. These include FPGA control to turn its power on and off, and to deploy different HWFs as they are required. This is difficult to compare with and without our system, as the support functionality is found throughout the middleware – both on the MCU and FPGA.

For example, switching the active configuration on an FPGA is done entirely transparently in our system (through an `ensureConfigurationPresent` function called implicitly with every stub), but requires a number of distinct functional steps. Not only does one need to manage and send the address where each configuration is stored in the non-volatile memory, loading a specific configuration from flash requires either a special FPGA configuration or MCU code that controls the booting through an interface like JTAG [262] (see Section 4.3.2).

7.2. Phase 2: Hardware Accelerator Optimisation

Creating an appropriate hardware accelerator for use in the Elastic Node platform can be broken up into two steps:

1. Create an architecture in an appropriate way to fit limited resource FPGAs, and

2. Optimise the design to improve performance and power efficiency.

By considering three HWFs that we created in Chapter 5 – namely the ANN, CNN, and IPCA – we have shown various ways that these two steps can be accomplished. All three of these examples were shown to be applicable to highly resource constrained devices when utilising the variety of optimisations presented.

What remains is to evaluate how well these optimisations worked based on our system and platform requirements. Specifically, we are addressing SREQ_{III}: Energy Efficiency by improving both the performance and energy usage of our accelerators, and SREQ_{I}: Local Intelligence through improved computational tools used to create a smart IoT device.

Due to the importance of correctly representing real numbers in hardware acceleration, Section 7.2.1 focusses on optimising fixed point representations. After that, the impacts of other optimisation techniques discussed are evaluated in Section 7.2.2. Next, the latency models we created are verified in Section 7.2.3 to ensure their accuracy and usefulness. Lastly, we revisit the use case presented in Section 2.1.1 to investigate the Elastic Node's capacity to perform facial detection on an autonomous drone or UAV.

7.2.1. Optimal Fixed Point Representation

The CNN HWF we created [35] is highly optimised for resource efficiency. This is partially due to the careful management of the fixed point representation ($\mathbf{Q}m.f$) used throughout the system. This provides developers using the HWF a configurable parameter for balancing accuracy with reduced resource usage, since larger number representations invariably require more FPGA resources.

The relationship between fixed point representation and CNN accelerator accuracy is shown in Figure 7.6, where we compute the model accuracy, recall, and F1 score (which combines the precision and recall) for each different representation. Particularly the number of bits used for the fractional part directly impacts the computed results of the CNN when compared to a full floating point representation. This also highlights the diminishing returns when using more than 6 bits for this particular dataset (the MIT-BIH Arrhythmia database [162]), suggesting that more than 8 bits would be wasteful.

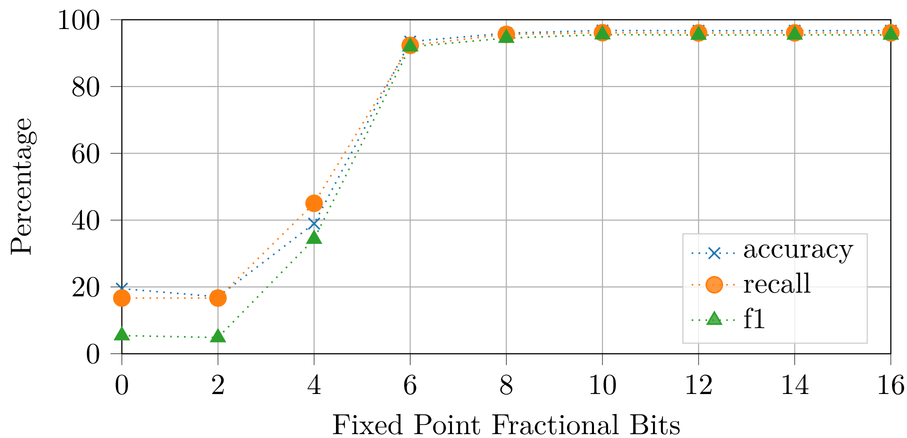


Figure 7.6.: Accuracy, recall, and F1 score for various fixed point representations

As highlighted in Section 5.3.2, the objective of this use case was to convert a standard TensorFlow CNN model to be compatible with the Elastic Node. The model in question consists of two 1D convolution layers (with 16 filters and a kernel size of 7) with max pooling layers in between. It also includes a global average pooling layer and a single dense layer for the classification output.

This is further investigated in Figures 7.7 and 7.8, where different resources' consumption are plotted as the fixed point representation is altered. This is done for three different FPGAs in the Spartan 7 family, showing how each device's resource consumption is affected.

An interesting behaviour is seen with the S15 for representations from **Q**14.28 to **Q**18.36, where the number of DSP slices plateau. This represents a numerical bitwidth of 42 to 54, which we believe can all fit into the same MAC computations available in the DSP slice of the Spartan 7 family.

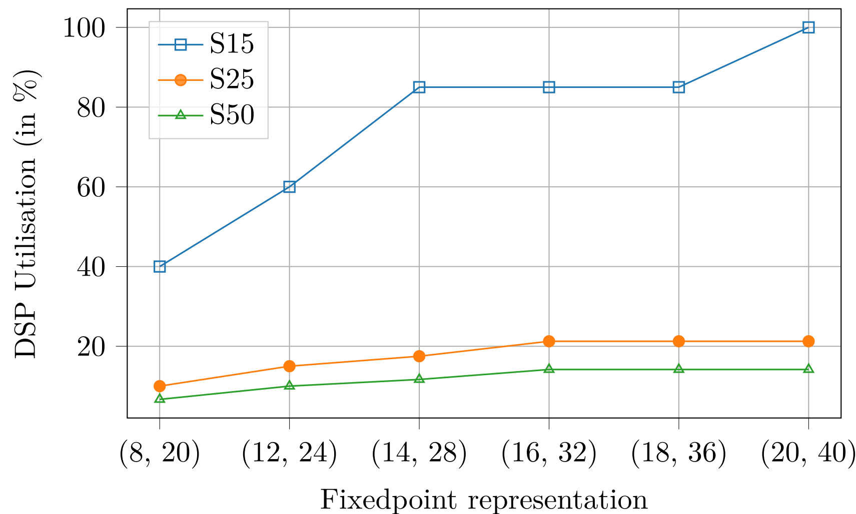


Figure 7.7.: Impact of fixed point representation on DSP consumption of various FPGAs

This type of behaviour can be very difficult to predict, since the synthesis tools perform a number of optimisation steps based on the mathematical function being implemented. For example, since each operation's resulting representation is the same as the input (instead of having both the number of fractional and integer doubled) the tools can discard a number of result bits – or in fact not compute them in the first place. This is both one of the more complex and beneficial parts of designing custom hardware architectures to solve mathematical problems – very fine-grained and difficult to understand optimisations can be made.

It should be noted how low these resource consumptions are for an accurate CNN hardware accelerator. This was possible due to the memory configuration previously mentioned, as well as architectural optimisations like combining multiple layers into the same pipeline, and fine-tuning the fixed point logic representation. Although it can be seen that the S15 relatively uses more than 4× as many DSP resources as the S25 or S50, comparisons between different devices is not particularly useful. These graphs should instead be used to:

1. Find the smallest FPGA where a specific parametrisation (fixedpoint representation) will fit in order to minimise power consumption and cost, or to
2. Study the trade-offs for a specific device between two parameters (i.e. what is the resource cost for moving from **Q**14.28 down to **Q**12.24 vs up to **Q**16.32) to fine-tune chosen parameters.

The resulting hardware accelerator design was used to create a highly accurate power estimation to further ease the development process. Each option is synthesised and implemented for differently sized FPGAs as shown in Figure 7.9.

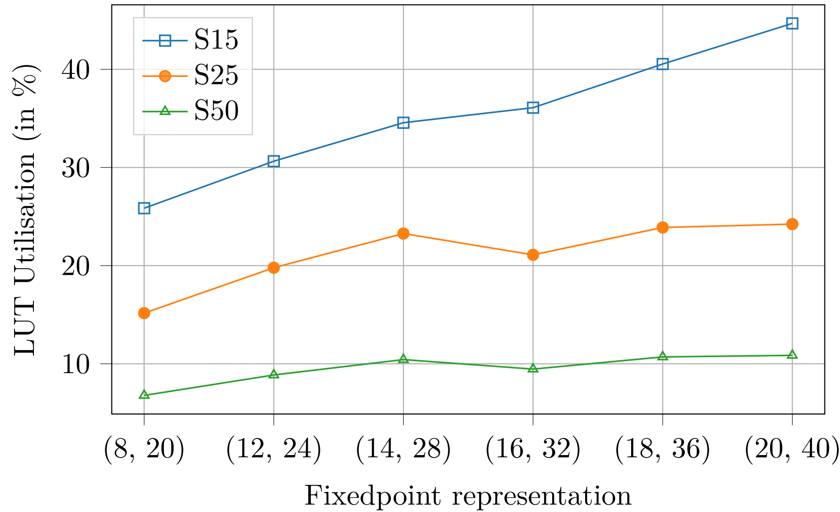


Figure 7.8.: Impact of fixed point representation on LUT usage of various FPGAs

These power estimates demonstrate the complex power characteristics of different FPGA models and configurations. In simple terms, the power usage of an FPGA consists of static and dynamic power [191]. The static power mostly depends on the FPGA model chosen, while the dynamic power is dependent on the configuration deployed in its PL.

Figure 7.9 demonstrates the cost of having larger fixed point representations. Since they require more logic and more DSP slices (each slice can only process a certain number of bits – in the case of Xilinx 7 series it can perform an 25×18 bits wide multiplication), the dynamic consumption of the resulting configuration increases almost linearly. When increasing the representation from **Q**8.20 to **Q**20.40 (28 to 60 bits total) the power increases linearly for 31 mW to 45 mW for the S15, while the S50 increases from 82mW to 93mW. It is interesting to note the decrease in power for the S15 and S25 from **Q**14.28 to **Q**16.32, which we hypothesise is due to simplified logic or routing due to how the mathematical operations fit into available DSP slices.

7.2.2. Common Hardware Design Techniques

In this section we evaluate the design optimisations presented in Section 5.2 as they contributed to making our IPCA HWF viable on low-powered reconfigurable hardware. This is done by evaluating the feasibility of the final design, and comparing it to other designs from literature (where these optimisations have not all been applied).

Resource Consumption To start, we study the resource consumption for designs of our IPCA HWF with various input matrix sizes. Table 7.4 presents this directly in the absolute numbers of resource units required for FPGAs from the 7 Series from

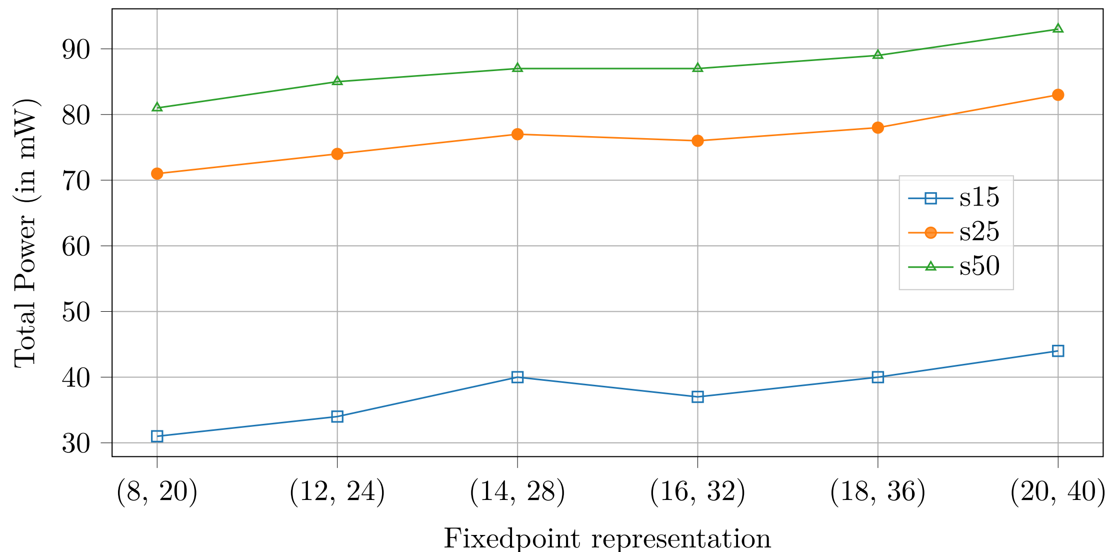


Figure 7.9.: Power consumption estimate for different fixed point representations on various Spartan 7 FPGAs

Xilinx. This is done since all FPGAs from this family offer identical resources in different quantities, providing a good indication on the device size required for each matrix size.

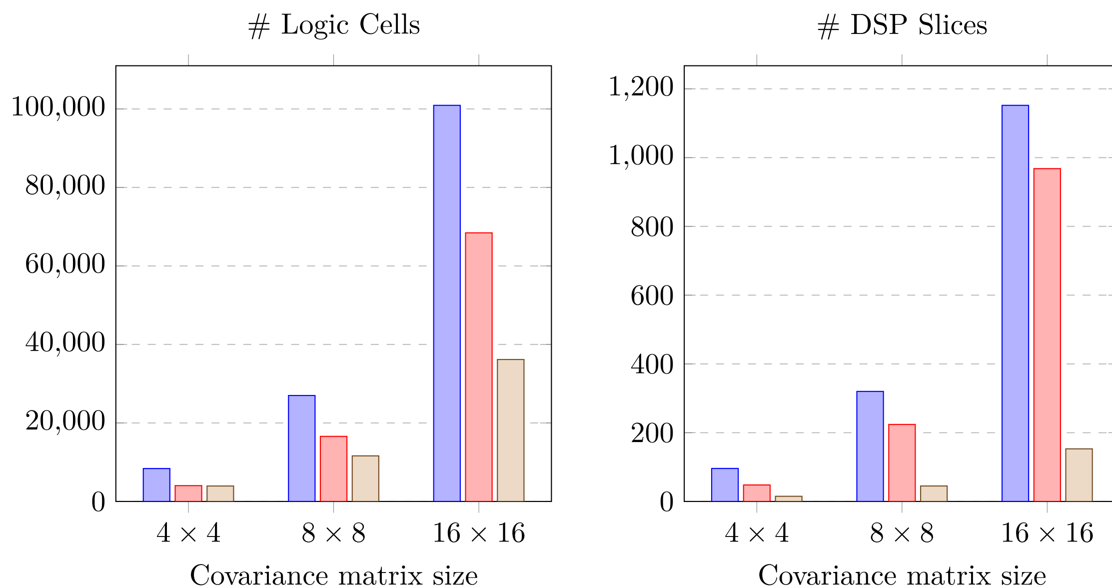
Table 7.4.: Synthesis results for Xilinx-7 Series FPGAs in absolute numbers

Matrix width	**Logic Cells**	**Flip-Flops**	**DSP Slices**
4×4	3,940	1,497	15
8×8	11,616	5,548	45
16×16	36,165	21,612	153

For example, the number of DSP slices required limits which FPGA is required to instantiate certain matrix sizes. The smallest Spartan 7 available (the S6) only offers 10 slices [266], and therefore would not support any of the options shown. However, the next size (the S15) offers 20 slices and therefore would support the 4×4 matrices, but not 8×8. This creates a simple first step mapping between application complexity (matrix size) and the size of FPGA required.

However, the absolute numbers of resources required in isolation can be difficult to interpret. Therefore, we present in Figure 7.10 the comparable numbers of other solutions from literature. The most relevant approaches that provided their resource and power usage numbers were those by Guerrero-Ramírez *et al.* [89] and Korat and Alimo-

hammad [121]. Both of these are hardware architectures designed for solving PCA using QR, but are limited by their dependence on the CORDIC algorithm.



[[Image Discarded Due To `/tikz/external/mode=list and make']]

Figure 7.10.: Comparison of the resource utilization of similar approaches in related work

The number of logic cells and DSP slices are compared in Figure 7.10, as we found them to commonly be the limiting resource on these small FPGAs. It is important to note that the two alternatives used for comparison are designed for considerably larger and more expensive FPGAs: the architecture from Guerrero-Ramírez *et al.* is aimed at the Altera Stratix IV[89], while the one from Korat and Alimohammad targeted the Zynq ZC702 [121].

In contrast to this, our design is specifically designed to operate on small and cheap embedded FPGA families such as the Spartan and Artix families. Therefore, Figure 7.10 highlights the drastically reduced resource requirements offered by our design. While the other solutions require around 1000 DSP slices to solve 16×16 matrices, ours manages with fewer than 200.

The power savings of using a smaller FPGA can be clearly seen in Table 7.5, where the implemented power costs for FPGAs from different Xilinx families are shown. All of these networks are aimed at 16×16 eigensolvers, offering between roughly 1.2 W and 1.4 W total device power draw.

The mild variations visible between them is due to the varying placement and routing of signals based on their different sizes, and each FPGA's specific overhead (static power usage). Although the other solutions from literature do not offer accurate power usage numbers, the FPGAs they use commonly require at least a few Watts for operation.

Table 7.5.: Implementation results for matrix size 16 × 16

Target	LUT	FF	DSP	Power [W]
Spartan-7 XC7S100	49%	17%	96%	1.402
Artix-7 XC7A100	49%	17%	64%	1.379
Artix-7 XC7A200	23%	8%	21%	1.238
Kintex-7 XC7K70	76%	26%	64%	1.425
Kintex-7 XC7K160	31%	11%	26%	1.214

Since our IPCA HWF design is supported by considerably smaller and lower power FPGAs, one can safely assume that we offer lower power consumption.

Performance The model for our IPCA HWF requires the clock frequency of the design, as well as the matrix width and the configurable iterations until convergence (which is primarily dependent on the size of the lookup table developed in Section 5.2.5). Therefore, we used the implemented FPGA architecture for every matrix size to retrieve the maximum possible clock speed on each FPGA.

The result is shown in Table 7.6, showing that the slowest maximum speed is from the Spartan FPGA (XC7S100) which offers 220 - 239 MHz. Higher end devices such as the Artix (XC7A100) or Kintex (XC7K70) are capable of higher rates, culminating in the 339 MHz possible when implementing the 4 × 4 matrix on the Kintex device. This increased clock offers 54% higher throughput than the Spartan FPGA, primarily because the Kintex family is designed for faster signal processing rather than lower cost.

Table 7.6.: Maximum operating frequencies in MHz depending on the matrix width

Target	f_{max} matrix width		
	4 × 4	8 × 8	16 × 16
XC7S100	239.01	228.31	219.11
XC7A100	265.75	237.87	237.98
XC7K70	339.90	272.18	252.46
EP4SGX230 [89]	235.32	220.15	201.35

To compute the latency for processing an incoming matrix, these frequencies are combined with Equation (5.18) from Section 5.2.6. This provides the time required to feed a matrix into our design, which both trains and queries the existing IPCA model. These

latencies are shown in Figure 7.11, where they are compared with the solution from Guerrero-Ramírez *et al.* [89] and a mainstream desktop CPU.

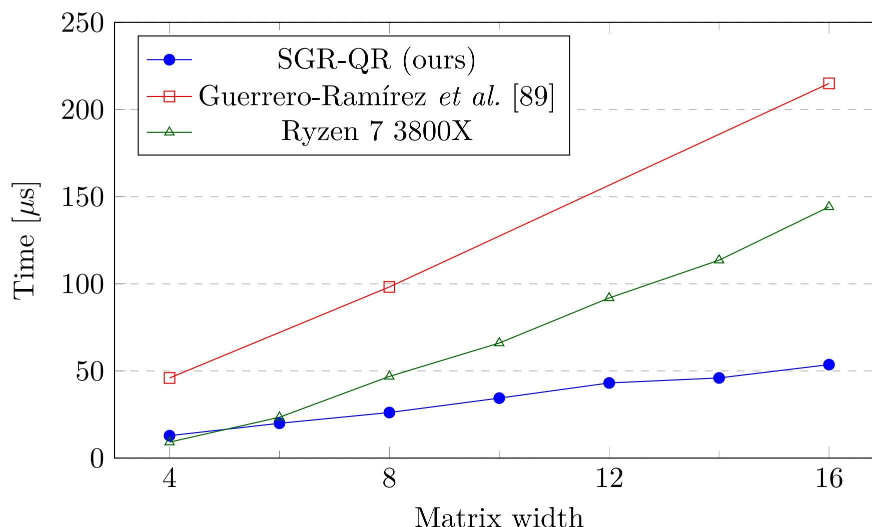


Figure 7.11.: Time in μs required to compute a single eigenpair of different matrix sizes, based on theoretical throughput of each solution (solutions/s).

The results shown in Figure 7.11 are based on each implementation's *solutions/s* metric, indicating how many full sets of eigenvalues and eigenvectors can be computed per second. For the solution from Guerrero-Ramírez *et al.*, this is based on the number of steps in the QR solution, the number of fractional bits computed, and the matrix width itself. This result is more linear than ours, as we also consider the maximum frequency possible for each matrix size. The software solution is measured experimentally using the LAPACK[1] eigensolver.

It is important to note here that the competing FPGA solution is again implemented on a Stratix IV, while ours is on the considerably lower end Spartan 7 FPGA. The CPU solution is operating on the 105 W Ryzen 3800X, but is severely limited since these solutions are commonly $O(n^3)$ while our solution is $O(n)$. Additionally, software solutions are commonly single-threaded [190].

This result highlights that although the resulting latency of all three these options increase linearly, the time required to process a single matrix on our system is consistently lower. It performs faster than the competing solution from Guerrero-Ramírez *et al.* for every width matrix, and is only outperformed by the sequential CPU-based solution for the smallest 4×4 matrices. As the matrix size increases, the improvement offered by our solution becomes more apparent, offering 16×16 matrices at a similar speed than the other FPGA solution processes 4×4 and the CPU solves 8×8.

[1] `http://www.netlib.org/lapack` (last visited: 2021-12-13)

This clearly shows the benefit of the hardware architecture design optimisations applied to our IPCA hardware accelerator. The drastically lowered resource requirements allow the usage of considerably smaller and lower power FPGAs, while the high computational performance allows it to outshine the competition.

7.2.3. CNN Latency Model Verification

Using the latency model developed in Section 5.2.6, we can calculate the processing speed and resulting latency for training and querying our CNN HWF. What remains at this stage is to verify that these models offer accurate estimates.

The inference latency formula created in Equation (5.4) can be used to estimate the time taken for performing a single query through the system. Since the design depends on the reuse of implemented neurons through on-demand loading of weights and biases, it cannot be effectively pipelined. Therefore, the latency scales linearly when doing multiple queries.

To verify our latency model is accurate, we deployed our CNN hardware architecture to the Elastic Node v4. Using the readily available clock sources on the board (32 MHz and 50 MHz), the comparison between the latency estimated using the model and the real-world experiments are shown in Figure 7.12.

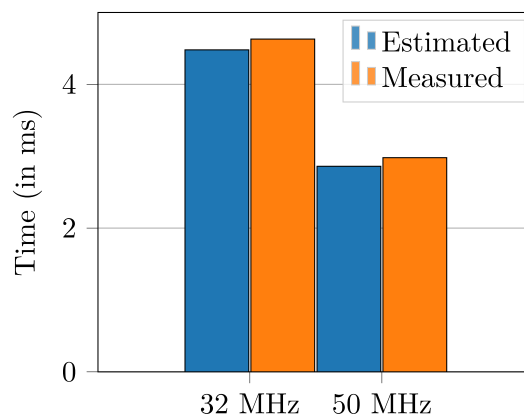


Figure 7.12.: Comparison of the latency model of our CNN HWF with real world experiments

This shows that the model can accurately predict the actual latency within a 4.2% error, which can be attributed to the overhead involved in exchanging data and commands with the architecture. The benefit of having such an accurate latency estimate materialises when designing for specific application requirements. Especially if this includes real-time deadlines, it allows the developer to easily compute the maximum network size that can be implemented given a specific clock rate, or alternatively sets a minimum clock requirement given an existing TensorFlow model that needs to be deployed - assisting the choice of FPGA used.

7.2.4. Incremental PCA Facial Detection Use Case

We also need to evaluate the validity of the real-world use case of facial detection on UAVs introduced in Section 5.3.3. When operating in a Fully Autonomous Aerial System (FAAS), they essentially act as data sources for the computation happening on edge and data centre servers [26]. Since they commonly operate under strict energy limitations due to their dependence on battery power for both processing and locomotion, their processing capabilities can be very limited.

Therefore, other approaches rely either exclusively on offloading [27] or on taking shortcuts to simplify the processing task. Some examples of this from literature is using downsampling ($5-12$ Frames Per Second (FPS)) and compressing tiny images (17 FPS) so they can be used as input for neural networks [24, 81]. These both carry the danger of losing critical information that only occur in small regions of the input images, but when the only alternative is offloading (which can be very energy expensive and take in the order of seconds [27]) it may be unavoidable.

Our solution [36] suggests an alternative approach, where key images are identified through the facial detection algorithm. This then acts as a preprocessing technique that reduces the number of images that need to be transmitted, or even reduces the sent data to only the areas of interest.

We found during our investigation that a 16×16 matrix width was adequate for performing facial detection on the well-known FDDB dataset [110]. Using a naive classifier, the accuracy was increased from 44.6% to 55.5% when increasing the size of the matrix from 4×4 to 16×16 (effectively increasing the number of eigenvectors considered). This is similar to the results of related approaches [110]). A sliding window of 250×250 is moved over the 640×480 input images. We found that 95% of the overall variance could be described using 62.5% of the eigenvectors.

The processing speed for training and querying using our IPCA HWF is shown in Figure 7.13, where the number of frames that our architecture can solve per second is shown against the size of the input matrix. This considers both the decreasing maximum clock frequency presented in Table 7.6 and the higher number of computations required for larger matrices.

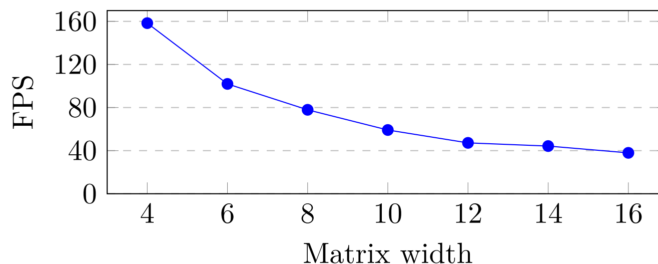


Figure 7.13.: Frames per second for facial detection application

This shows that even though the processing speed is predictably reduced when increas-

ing the matrix size, the processing speed remains above 30 fps. This relation between speed and computational complexity can be combined with the device power usage from Table 7.5 to choose the most appropriate device for a specific set of requirements.

Using this information, we can estimate that our system would consume between 3.14 μJ for $n = 4$ and 68.61 μJ for $n = 16$. Considering that transmitting even just an image preview can take a UAV 1.4 s [27], we can safely assert that ours would be a more energy efficient solution (as related work does not offer comparative numbers).

7.3. Phase 3: Intelligent Cooperating Devices

The next phase of this work involved the design and implementation of an AI that optimises device behaviour (as set out in SREQ_{II}: Automated Cooperation Optimisation). This focussed specifically on an augmented offloading problem which considered sending incoming jobs to peers, batching them locally for increased efficiency, or deploying the required FPGA configuration locally and beginning processing.

Initial experiments have been done to demonstrate the usefulness of our system, to find its current limitations, and to control whether our requirements set out in Section 6.3 have been met. This was done in simulation, based on the analytical model set out in Section 6.4, as well as performance and power consumption data collected from the Elastic Node platform as described in Section 4.5.2.

The experimentation procedure is set out in Section 7.3.1, followed by a set of experiments originally presented at ACSOS 2020 [34]. This starts with a comparison of the agents developed in Section 7.3.2, followed by a study on the limitations of the system state representation, and more complex experiments with cooperating devices under catastrophic failure in Section 7.3.4.

After this, the SRL agent is introduced and compared to the Q-table ones in Section 7.3.6, followed by a similar experiment for dynamic environments in Section 7.3.7. Finally, we conclude with some thoughts on the evaluation in Section 7.4.

7.3.1. Experimental Setup

The experiments detailed below involve a number of devices, and either a single shared or multiple device-local agents. When the scenario is centralised, there is a single agent which is trained on data from all of the devices. This is done primarily as a simplifying first step, temporarily ignoring our requirement for decentralisation set out in IREQ_{III}: Realistic Implementation.

In contrast to this, the decentralised version of the system treats each device as independent with its own agent. These agents are trained only from data available to that device, and only learns from the experiences of that single device. This is to simulate a system where each device truly operates independently, minimising the communication and cooperation overhead.

Although a hybrid solution can be envisioned that routinely shares experiences between different agents, this still increases communication overhead when compared to a fully decentralised solution. Instead, we envision our agent being self-sufficient and able to learn by itself. This creates a much more scalable system where each device can be deployed independently – learning as it goes along.

7.3.1.1. Experiment Runtime

To evaluate a number of different scenarios and situations, we performed time-series simulations of these devices. Based on the scenario description, jobs were dynamically created at the different devices – either randomly or at a fixed interval. As described in the Analytical Model in Section 6.4, performing each job involves doing a number of subtasks that need to be completed. The durations and power consumption of each subtask was also modelled using a Gaussian distribution.

Performing these time-series simulations is computationally intense, requiring thousands of discrete time steps to be simulated for even a small simulated battery. Add to this the need to repeat each scenario multiple times to ensure repeatability and to mitigate outliers, as well as needing to vary multiple parameters to create a meaningful comparison. This created the need for a convenient and efficient simulation framework.

To this end, a tool was developed in Python that can simulate many scenarios in parallel. As simulating each of these scenarios generally only utilises one logical system core, a multithreading scheme was implemented using the Multiprocessing[2] library. Although creating a separate process for each of them adds some overhead when launching many at the same time (and memory overhead for storing the needed variables), the advantage is having entirely independent processes that each have a local copy of important records and objects. This makes it considerably easier and safer to alter 'local' variables in memory without the fear of affecting other simulations happening in the same system.

Performing the experiments involved a large number of these simulations. Therefore, each was set up as a Docker image and deployed to one of our servers using Kubernetes[3]. This allowed not only a convenient way of deploying a specific experiment to run on the processing power available, but introduced a powerful way to monitor and retrieve results from these simulations.

Using these techniques, each of these experiments are repeated 128 times, and the results are provided as the average for all agents involved over these runs. Unless mentioned otherwise, these graphs also show the error bars to indicate the variance between the recorded runs. The null hypothesis testing involved here is focussed on whether the relevant intelligence requirement has been met, i.e. testing whether our introduced intelligence has had the effect that we aimed for. Therefore, the analysis that follows is a mixture of exploratory verification that the system functions as designed, and an evaluation of the requirements set out in Section 6.3 – both qualitatively and quantitatively.

[2]`https://docs.python.org/3/library/multiprocessing.html` (last visited: 2021-12-13)
[3]`https://kubernetes.io` (last visited: 2021-12-13)

7.3.1.2. System Parameters

Each of the experiments were done with a job rate that is a regular interval T. Since this is the overall rate for entire system, it is shared by all of the devices. This ensures that the system as a whole always has access to the same number of jobs in a given time span. Varying the exact rate at which jobs are created has the effect of bounding where the learning algorithm can provide a benefit. For example, if jobs arrive too fast the devices benefit less from batching as their FPGAs are all active.

When overall jobs accomplished is used as a metric, it is important to make the performance of each represented system comparable. Therefore, a completed job is defined as one that is created, moved to the relevant processing device (offloaded to a peer in some cases), fully processed, and has its result sent back wirelessly to the device that created it.

Each device is accurately represented through empirical power and subtask duration measurements from the Elastic Node v4. Using the power monitoring available on the board, each component's power usage is characterised for various workloads. A Gaussian distribution is fit to each combination of hardware component and subtask, allowing us to accurately estimate the overall power cost of each task/subtask. Again this is used to model sensor noise due to the central limit theorem [84] suggesting that continuously sampling this independent random variable will tend to a Gaussian distribution.

By simulating each device's activity with such high fidelity, a detailed energy usage can be found. Note that the simulation model discussed here is not used for decision making at runtime by the devices – instead only used at design time to train the agent model. Additionally, the duration of each subtask is characterised relative to the device's operating frequency. This is combined with the device model for improved flexibility.

7.3.2. Agent Comparison

The first experiment targets the question 'can an effective agent be designed by defining its *action space*, *system state*, and *reward function* in this way?'. To investigate this, two agents defined in Section 6.5.4 are created with different objectives. The "basic" and "lazy" agents are used to show how the reward function can be used to implement a developer's *intent* for the devices. This directly addresses IREQ_{II}: Generalisability, allowing a user to mold the design of an agent to their application's needs. It also addresses IREQ_{I}: Rationality, showing how reasonably intelligent behaviour can be created by defining the agent's intent.

The basic agent uses the full reward function defined in Equation (6.6) so that its reward function becomes

$$r_b \triangleq R_j + R_e + R_d \tag{7.1}$$

and includes all our reward components. The lazy agent uses a reduced version

$$r_l \triangleq R_e + R_d \tag{7.2}$$

which completely ignores the reward for completing jobs. Instead, the lazy agent is only interested in minimising its energy usage and avoiding the terminal energy state.

The other differing parameter between the agents shown in Figure 7.14 is whether they learn centralised or decentralised. For each learning agent, both variations are shown to illustrate differences that appear between their performance or behaviour. Although one would expect them to learn the same behaviour, there would likely be a difference in how quickly they learn as well as their learning stability. By definition, an agent only learning from a single device's experiences would have less input per episode and therefore perform less state exploration.

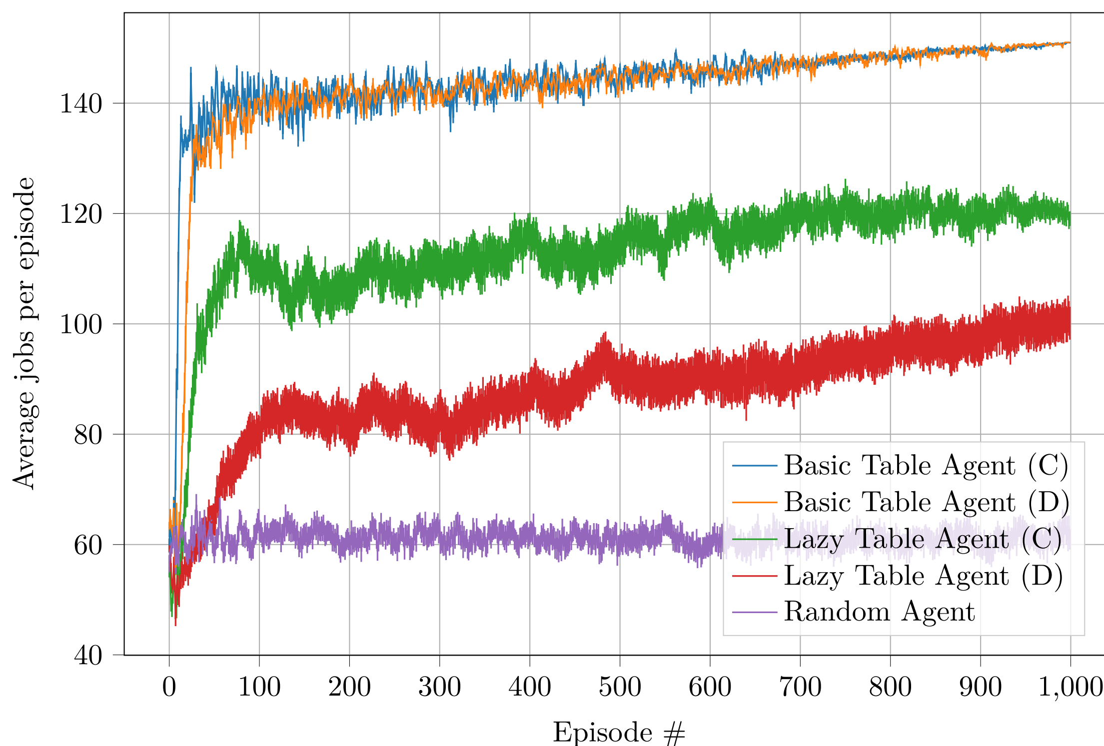


Figure 7.14.: Agent comparison between (C)entralised and (D)ecentralised basic and lazy agents, showing the relative performance (number of jobs performed per episode) of each

As a comparison, we also show a random agent that does not perform any learning. Instead, it chooses which action to perform completely at random. Apart from highlighting the need for careful decision making on these devices, it also creates a performance baseline that represents practically a worst case scenario in terms of rational behaviour [206].

Figure 7.14 provides the average performance and error bars of these 5 different agent types. Using the average number of jobs completed by the two devices during that

episode as a metric, the basic agent is shown to greatly outperform the other agents. Interestingly, there is no real statistical difference between the performance of the centralised and decentralised basic agents. This shows that this agent is very capable of learning the "correct" behaviour by itself, without requiring centralised knowledge.

The learning agents all utilise a decaying epsilon-greedy that reduces ϵ over 1000 episodes from an initial 0.1. Using a larger initial ϵ increases the variance in performance between different runs, while a slower decrease extends the agent's ability to escape local minima and improve its behaviour. The Q agents all use a γ of $1 \times 10^{-}3$ and a learning rate of 0.1, which were chosen empirically based on brief tweaking of the designed agent behaviour.

The lazy agent is shown to perform considerably fewer jobs over its lifespan than the basic agent, clearly demonstrating that it is learning to target a different goal. Additionally, it is very interesting to note that the centralised version of this agent reaches its steady state noticeably faster than the decentralised one. In contrast to the basic agent, the lazy agent seems to benefit greatly from having access to training data from both devices.

For another insight into the behaviour this agent learns, we visualise in Figure 7.15 its Q-table after learning for 1000 episodes. Each of the system states is clarified on the left, as the table gives a coloured representation for the value of each discrete state. The expected value for each action from that state is then represented using a colour-coded system where red is very negative, blue is very positive, and black remains unknown.

We can see that in this case each learned state is negative to some extent, showing that the negative influences of the energy and death rewards outweigh the job rewards. However, the exact values of these expected values are irrelevant, as only the relative values for each state are used for decision-making. We also see that some states are unexplored – either because they are exceedingly unlikely or because they are impossible due to heuristically defined behaviours. For example, for each state where there are 5 jobs in the queue already, this agent has reached its maximum job queue size and is forced to start locally processing. Therefore, only the *local* action has been explored at this state.

7.3.3. System State Limitations

This heuristic and practical limitation limiting the behavioural scope of the agent urged us to investigate what happens when the system state is expanded. Apart from expanding the search space that the agent needs to explore, the agent is afforded greater freedom to perform actions that previously were not possible. In the extreme case – where there is no queue limitation and devices can simply store jobs infinitely – a lazy agent might choose to never perform any jobs (due to the energy cost incurred).

An experiment was performed that studies the variations possible with expansion of the system state. Again two devices were assigned similar but separate agents in each scenario, learning decentralised and independently. Since here we are interested in the

Figure 7.15.: Basic agent Q-table

final policy π learnt by each agent (and not the learning behaviour over time), each agent is trained for 1000 episodes, after which they were switched from 'learning' phase to 'production'. At this point, all learning on the devices is disabled to represent a system where devices learn ideal behaviour over a period of time before being deployed in a 'final' stable state. The devices were then allowed to follow a pure greedy policy in which they selected the available action with the highest Q-table score for 10 episodes, and their average performance was taken as one data point. This was again repeated for stability and to find statistically significant differences.

Each different agent's system state was changed by increasing its maximum job queue

size. For obvious reasons this is only done for the learning agents, which is why only the basic and lazy agents are shown in Figure 7.16. Similarly to Figure 7.14, the agents are shown to perform very similarly when the maximum queue size is small. However, as the queue size is expanded, the lazy agent is able to deviate more from the basic agent until it performs no jobs. At this point when the maximum allowed queue size is around 100, the device governed by the lazy agent can simply avoid doing any and all work in order to extend its own life – with no regard for the amount of work accomplished.

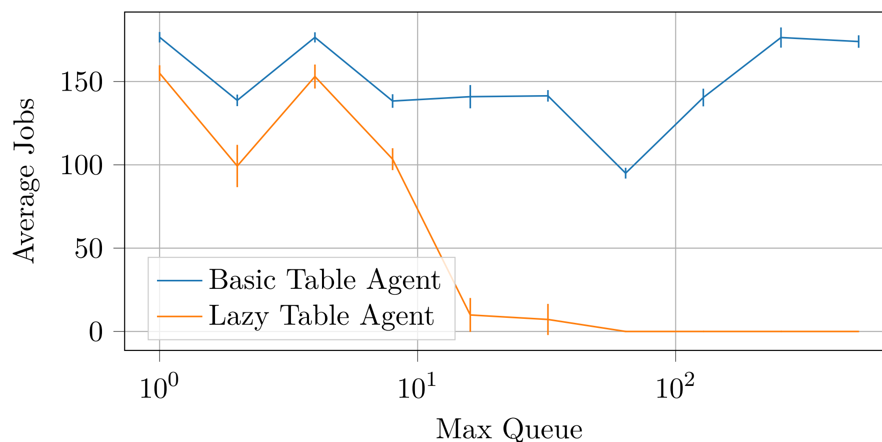


Figure 7.16.: Impact of system state expansion

Although this behaviour of the lazy agent might seem extreme, it shows the potential of designing agents in this way. For example, the developer might train both of these agents and switch a device from one policy to another as the user's requirements change. This could be useful when the survival of a specific device is deemed essential and it is crucial for it to survive as long as possible. For example, consider a drone that has managed to get to a hard to reach position that another drone cannot easily reach to relieve it of its duties. In that case, that drone should focus simply on data acquisition instead of processing.

The substantial changes in agent behaviour and performance with altering the system state raises another issue. This dependence highlights the importance not only of correctly modelling the system state, but also the need to accurately store the state-action values. In the case of a Q-table, this leads to the concern of exponentially growing state space, as each additional binary state doubles the total number of discrete states. One approach to addressing this recently has been used in Deep Q [159] and Deep Reinforcement Learning [105, 135, 271], where a deep neural network is used instead of a table. Implementation of such a system differs, however, and no common understanding for what exactly the role of the neural network is has been established. In some cases, the neural network's output is used directly as Q values, while others use a more complex version where sequences of outputs are used together to form an offloading schedule [105].

7.3.4. Catastrophic Failure

To satisfy IREQ_{IV}: Dealing with catastrophic failure, we need to test how the system responds to having some devices fail. In an ever-changing dynamic system – as many IoT and smart city applications are – one cannot assume that all devices will remain active or reachable. Therefore, it is critical that remaining devices (and the system as a whole) can stay functional and still accomplish their goals.

Instinctively one would hope that the remaining agents would adopt the workloads of the devices that were lost, analogous to biological systems like ants and bees [118]. These systems are very sensitive to environmental changes, and the organisms must adapt to ensure the success of the group. Ant colonies, for example, have shown increases in specialisation (i.e. greater tendencies for more ants to dedicate themselves to a specific role) to satisfy the changing needs of the colony as a whole. More of them might choose to focus on hauling food when a large deposit is found, reducing focus on secondary jobs such as nest cleaning.

A useful metric for monitoring this type of behaviour is the Division of Labour (DOL) as coined by Gorelick *et al.* [86]. This metric can indicate the rise and fall of agent specialisation over time, providing some insight into how agents dedicate to certain tasks over time. It combines the average individual agent entropy with the mutual entropy of the agent and the rest of the population. By populating an $n \times m$ matrix with a agent-task value for each agent (n) and task (m) combination, the tendency for that agent to do each task can be captured.

At the end of each episode the matrix is normalised and Shannon's entropy is used to calculate each agent's entropy score across all possible tasks. The mutual entropy is calculated by evaluating

$$I(X,Y) = \sum_{x \in X, y \in Y} p(x,y) log \frac{p(x,y)}{p(x)p(y)} \tag{7.3}$$

across all agents in X and tasks in Y. Shannon's index (used to quantify diversity)

$$H(X) = -\sum_{i} p_i \log_2 p_i \tag{7.4}$$

utilises p_i the number of agents assigned to task i divided by the total number of applicable agents. Dividing Equation(7.3) by Shannon's index

$$D_{y|x} = \frac{I(X,Y)}{H(X)} \tag{7.5}$$

yields the DOL score along the interval [0,1]. Low scores directly relate to a low DOL across the population, relating to low specialisation between different agents.

In the realm of learning agents King and Peterson [118] have showed that catastrophic failure can create an opportunistic situation for increased specialisation in artificial systems. Extending on this idea, we created an experiment where different sized groups of

decentralised but homogeneous agents are deployed together in an environment for 1000 episodes. At the halfway mark (after 500 episodes), half of these agents are removed from the system to emulate a catastrophic failure in a real world experiment. Analogous to this, imagine a swarm of drones that are cooperating and learning to accomplish a shared goal. At some point in time, half of these drones are instantly disabled or removed. Thereby, an extreme case of catastrophic failure is created in order to evaluate the behaviour of agents under loss.

Adding another dimension to our experiment shown in Figures 7.17 and 7.18 is that it includes numerous different computational tasks (T_n), where each requires a distinct hardware accelerator to be deployed to the FPGA. In terms of the batching behaviour (see Section 3.2) this means that a number of distinct batches need to be created on each device, since each can only consist of jobs that rely on that specific computational task. Having a variety of different jobs is essential for us to accurately estimate the DOL, as it is a requirement for specialisation.

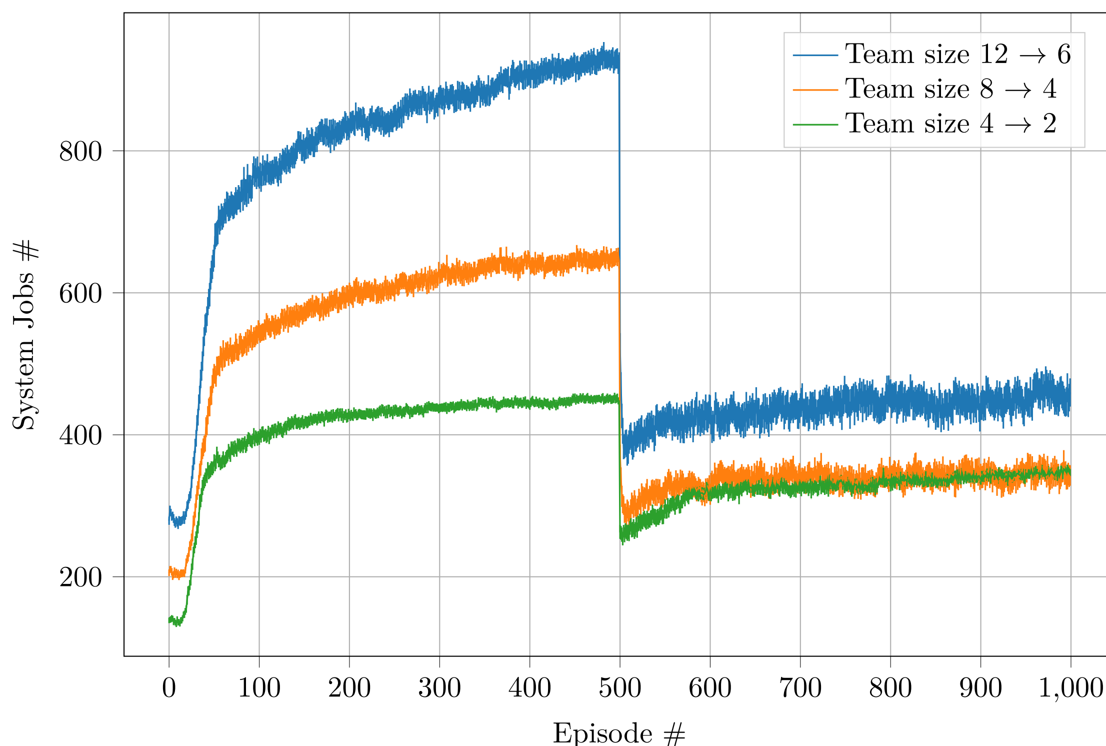


Figure 7.17.: Various sized (initially 4, 8, 12) teams cooperating to perform jobs. After 500 episodes half of each team is removed, and each team's performance (jobs per episode) is monitored. Each team can be seen to somewhat improve their performance after catastrophic failure.

This experiment shows that during the first half of the experiment each group of devices is steadily learning to improve their performance in both types of available computational tasks (i.e. performing more jobs per episode). During this period, the DOL is shown in Figure 7.18 to steadily decrease as the agents learn an even distribution of work between them.

The abrupt change that we introduced at episode 500 clearly has a strong impact both on the overall system jobs and the DOL. As can be expected, in each case half the devices are performing a substantial amount fewer jobs over the space of an episode. It is interesting to note that 4 devices halved to 2 seem to perform better than the other teams, reducing their accomplished jobs by roughly 25% while the other teams reduce by around 50%. This is likely related to the shared job rate, as devices will find it easier to batch more devices without reliance on offloading to peers when fewer devices exist.

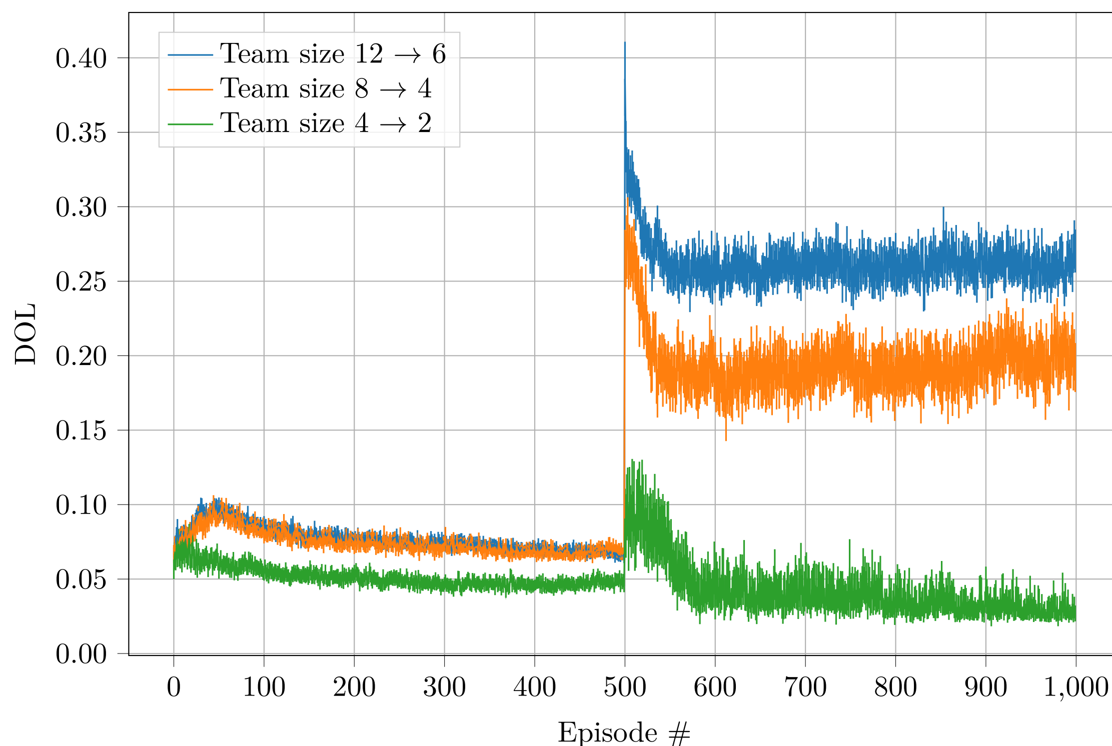


Figure 7.18.: The same experiment as before, monitoring the division of labour (DOL) for each team before and after episode 500. When half of each team is removed, each team's DOL drastically increases and then settles again as they relearn a new policy. The two larger teams stabilise at a considerably higher DOL while the smallest team (initially 4) ends up less specialised.

The DOL metric shown in Figure 7.18 tells an interesting story. Initially the three teams are seemingly behaving fairly similarly, without any statistically significant dif-

ference between them. Once half the devices are purged from the system at the halfway point the difference between the different teams is much more stark, indicating significantly different behaviour between the two larger teams of initially 12 and 8 devices, and the smaller team that started with 4. This suggests that the larger teams tend to show greater specialisation at this point, while the smaller team is left with only two devices and therefore reduces the DOL score even lower than before the purge.

These results support our hypothesis that these devices can respond to catastrophic change by increasing their individual specialisation. Instead of continuing as they were before, the new teams increase their tendency to focus on a specific task in order to improve the overall performance of the system. Another, possibly more significant result is that the devices show a new ability to relearn old behaviour. Note the curved increase in overall jobs performed after half the devices are removed, improving on the previously learned behaviour. One would rightfully expect that the old policy is not ideal for this drastically new environment, and it is encouraging to see that the system is capable of relearning its policy.

7.3.5. Heterogeneous Agents

All the experiments up to this point have been with homogeneous agents — either all learning together on one Q-table in a centralised fashion or each learning individually on their own Q-table when operating decentralised. By providing them with the same reward function and system state, the user can encourage them all to the same high-level goals – possibly making it easier for the agents to work together as their behaviour is easier to predict. Logically one would also expect them all to tend to similar Q-tables, as all the agents are defined in the same way and share the same environment.

This might not be the ideal in a real-world scenario, as agent heterogeneity has the potential to not only perform better but also to enable more sophisticated system designs where the user has improved control over the system behaviour. More specifically, King *et al.* showed that in some cases heterogeneous teams can outperform homogeneous ones when addressing a variety of tasks [119]. Interestingly, they also showed that heterogeneous teams do not win by default, as the composition of the team directly impacted the performance of the team as a whole. In some cases it was better to have a team of identical agents, while in others it was important to achieve the right balance between the different agents.

With this in mind, we decided to investigate the performance of different combinations of agents. In a real-world experiment, one scenario where this might make sense is with a team that consists of two different drones with their own agents: one that has a special sensor such as Light Imaging, Detection and Ranging (LIDAR) on it, and one that has a larger battery for more energy-intensive computations. It would make sense that the rational policy learnt by these two teams would be different, as they have different things to offer the system as a whole.

We evaluated the impact of agent heterogeneity by creating a number of teams with

different compositions of agents. In each scenario, a team consists of 10 agents, some of which are our basic agents while the rest are the random agents from Experiment 1 that do not learn. The main goal of this experiment shown in Figure 7.19 is to see what impact the introduction of these unpredictable random agents has on the learning and eventual steady state performance of our decentralised basic agents. Our chosen metric for this is the average number of jobs performed by basic agents per episode, which makes it trivial to compare the performance of the different teams.

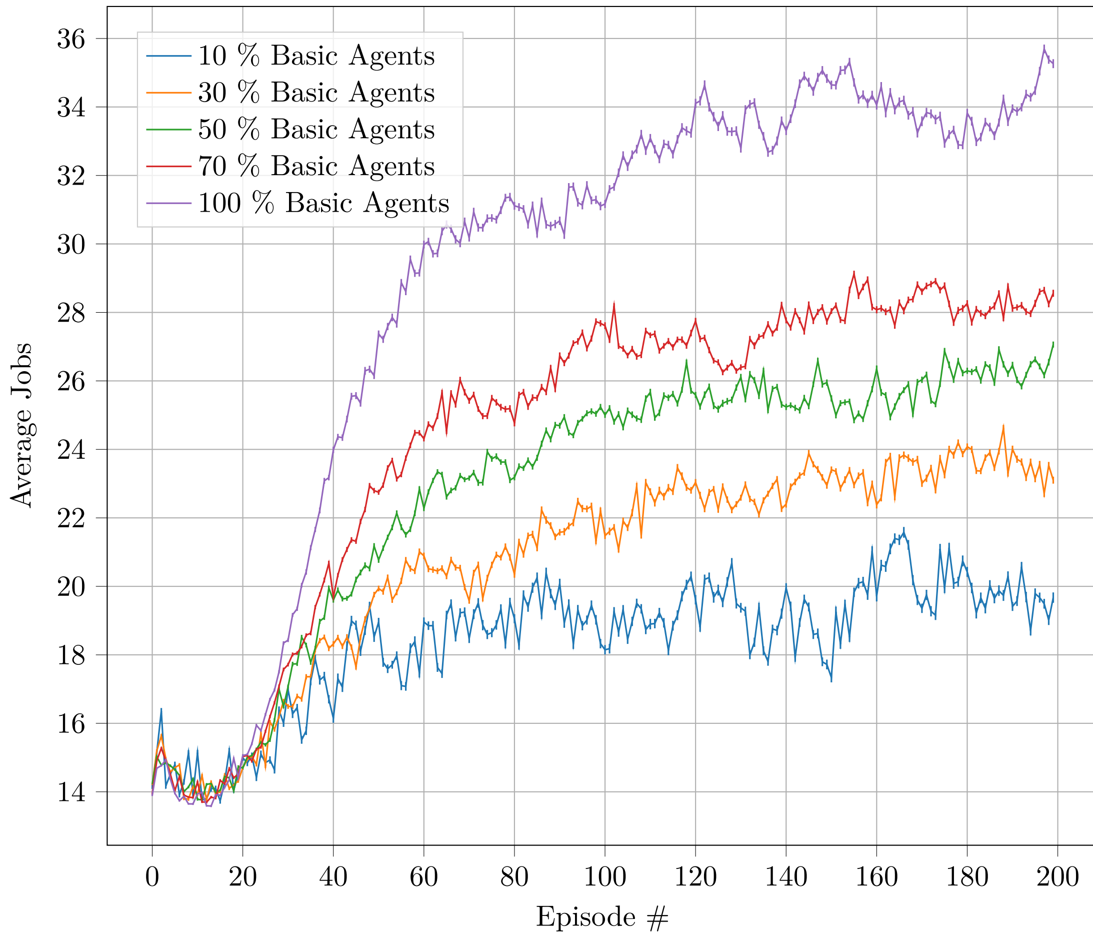


Figure 7.19.: Experiment studying heterogeneity in colocated agents, mixing different concentrations of basic agents colocated with random agents that do not learn. Higher percentages of basic agents are more efficient (when considering average jobs performed only by the basic agents), performing 75% more jobs per episode when only basic agents are present.

Thereby, the performance of each team is shown as a somewhat mixed team of 10 agents are co-located in an environment. For each team, a different percentage of them

are given the reward function and system state from the basic agent – ranging from 10% to 100%. It is clear that in this case the introduction of random agents to the team negatively affects both the performance and learning rate of the basic agents, culminating with 10% basic when there is only a single basic agent. In this case, the agent requires roughly twice as many episodes to reach its steady state and has their average number of jobs reduced by 57%.

This result clearly shows that the agent is capable of learning how to improve its performance even in the presence of unpredictable and wildly differently acting other agents. However, its ability to achieve this goal is negatively affected, suggesting that some measure of cooperation emerges from the more homogeneous teams that benefits the system as a whole. This is reminiscent of socially situated agents [14], since each agent has to manage its own (possibly conflicting) objectives. In our case this manifests in the random agents not sharing jobs as proficiently as the basic agents, reducing their ability to explore their search space and thereby slowing down their learning. This result also suggests that when agents are truly antagonistic with competing objectives, their ability to perform will be further negatively impacted.

7.3.6. Comparing Q-table and SRL Agents

When the storage of state-actions in a Q-table is no longer sufficient – e.g. due to growing system states or action spaces – alternatives need to be found. The first fundamental question to be answered in this experiment is if the modern adage that every problem can be solved with a neural network is true for this case. To answer this, we compare the performance of our existing Q-table agents with an SRL one that employs an SNN (as was explained in Section 3.5.4.2). The basic agent developed in previous experiments is used from this point onwards to represent the Q-table agent, as it performed the best. Due to the complexity and overhead in designing neural networks, one would hope that the SRL agent performs better than the Q-table one, or at least provides an advantage in some situations.

This experiment aims to compare the baseline performance of the Q-table and SRL agents for both pre-trained and untrained scenarios. Pre-trained agents are put through an offline learning process beforehand, and then their knowledge is transferred from the top performing previous agent to the new agents. In the case of the SRL agents, the offline training is done from a pre-trained Q-table agent that has captured the full set of available state-action values. The training data is then generated by creating a greedy state-action classification for each seen state. In contrast to this, a pre-trained Q-table agent simply copies the donor Q-table directly.

Additionally, an ensemble-like methodology is used for both types of agents, where the best networks are chosen from each agent to represent the expected performance. Since the Q-table agents are all initialised the same way, their performance is generally very similar. Due to the factors discussed in Section 6.5.1.2, the SRL agents are more dependent on their initial weights. This highlights a common shortcoming of neural

networks: one can have an appropriate network and an adequate training set but still not reach the wanted performance – simply because the randomised initial weights are incompatible.

The result of our initial comparative experiments can be seen in Figure 7.20, plotting the number of jobs per episode for both pre-trained and untrained versions of the Q-table and SRL agents. As can be expected, pre-trained agents achieve a new steady-state very quickly since they have already 'seen' most of the possible states. Both the untrained agents also manage to learn their environment, stabilising at roughly the same performance. Interestingly the SRL agent reaches it slightly faster, likely because it quickly learns some generalisations instead of having to learn individual states.

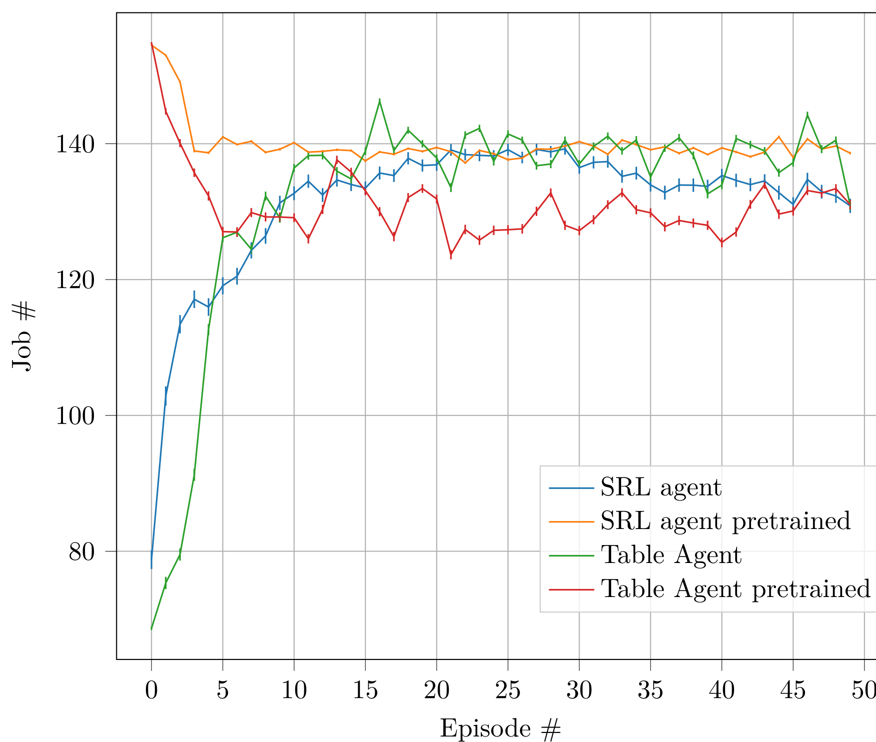


Figure 7.20.: Comparison of SRL and Q-table agents, both pre-trained and newly initialized policies. Although pre-trained agents start at more jobs than the newly initialised agents, all agents end up performing similarly.

This experiments serves to prove three fundamentals: firstly, the SRL agents are able to learn from a new initialisation directly in their environment, effectively learning the same level of performance as their Q-table counterpart. Secondly, it proves that the SRL agent can be pre-trained from a Q-table agent's policy. This should simplify a real deployment significantly, allowing the user to train one device either in simulation or in the field and then initiate all subsequent SRL or Q-table agents with that knowledge.

Thirdly, it shows that traditional techniques still prove to be very effective when compared to neural network-based solutions. The Q-table solution has its own shortcomings, but is shown to perform very similarly to the arguably more complex SRL agent.

One interesting peculiarity shown in Figure 7.20 is that both the SRL agents provide a more stable result than their Q-table variants. We suspect this is related to them being able to provide a superior initial guess for previously unseen (or untrained) states-actions. Whenever a Q-table agent reaches a previously unknown decision point, it is reduced to a random guess and needs multiple iterations before a definitive decision can be learnt. This inevitably reduces its transient performance slightly, since it is likely not going to make the correct decision. In contrast to this, the SRL agent can better generalise from existing information and make a more 'educated guess'.

7.3.7. Q-table vs SRL Agents in Dynamic Environments

To expand on this, we chose to experiment with a known shortcoming of the Q-table agent: in an unknown state, it is forced to guess entirely randomly. This behaviour is exasperated in dynamic environments where entirely unexpected situations can manifest. In this case, the agent is forced to expand its Q-table to accommodate these new state-actions – increasing its memory footprint and introducing a slurry of new unknown states. One would expect this to reduce its transient performance, creating an opportunity for SRL agents to justify their increased design complexity and other shortcomings. To amplify this contrast, the SRL agents retain the same architecture throughout this experiment – relying on their ability to generalise the situation.

In this experiment, teams of either Q-table or SRL agents are deployed to the same environment – all entirely untrained. They are then presented with 'difficult' jobs to perform for 1000 episodes, which require a longer time to process (i.e. costs more device energy). After this, they also receive 'medium-high' difficulty jobs which are slightly easier to process than the difficult ones. At this point (for 1000 episodes) they receive 'difficult' and 'medium-high' jobs with equal probability. At 2000 episodes, easier jobs are introduced in the same way, and at 3000 the easiest jobs too. This means that for the last 1000 episodes of the experiments, any agent can receive 'difficult', 'medium-high', 'medium', or 'easy' jobs with equal likelihood.

The objective of this is to test how the agents respond to previously unseen decisions. Note that one would expect any agent to perform more jobs per episode at the end of the experiment than at the beginning, since the average job it is provided with will be easier. Instead of always receiving difficult jobs, a mixture of jobs arrive – at worst as difficult as at the start of the experiment.

Figure 7.21 initially shows a similar result to Figure 7.20, where the SRL agent again learns its behaviour faster than the Q-table one. The real benefit of the SRL agent becomes clear in the transitional periods after new types of jobs are introduced. Every time, the Q-table suffers for a number of episodes with considerably reduced performance – losing 30-40% of jobs performed per episode. Invariably, the Q-table agent does recover

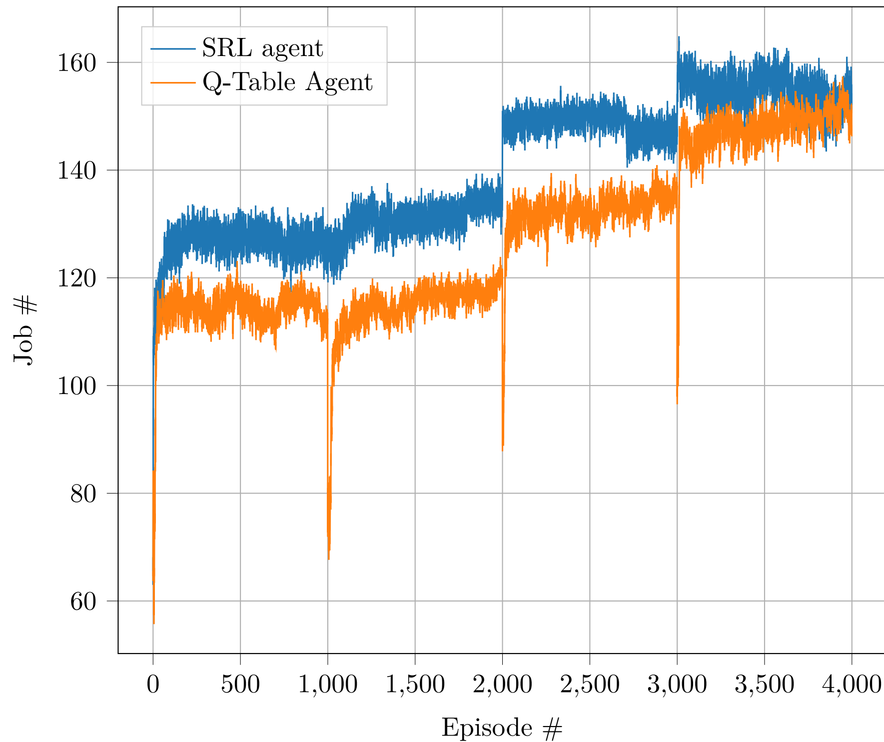


Figure 7.21.: Performance of SRL and Q-table agents under introduction of new tasks, showing that the SRL agent increases their performance smoothly and immediately, while the Q-table agent performs roughly 33% worse for a few episodes before recovering their policy performance.

its performance, but in some cases a number of episodes with lowered performance may be unwanted. Depending on the application, this may cause system outages as the devices need to be reinitialised or recharged. Towards the end of the experiment (after adequate training at the final scenario) both agents achieve basically identical results.

It is important to consider the production environment when choosing between the Q-table and SRL agents. In increasingly dynamic environments, the Q-table agent might grow its memory footprint beyond local capabilities. For smaller state-action spaces, the SRL agent might be overkill and increase the development complexity. Additionally, they are more susceptible to getting stuck in local minima – relying on ensemble systems to ensure overall system performance. Simply put, neural network-based agents can provide a worthwhile option by better generalising their learned knowledge. However, they are not the only way to solve problems – reminding of the old adage “when you only have a hammer everything starts to look like a nail”.

That concludes our presentation of our device learning AI, capable of governing an

FPGA-augmented smart IoT device's behaviour through RL. We feel it addresses our requirements for the agent as presented in Section 6.3:

I **Rational behaviour**: All of the presented learning agents (lazy, basic, SRL) have been shown to be capable of rational behaviour, choosing actions based on high-level objectives.

II **Generalisability to Various Applications**: The agent design was proven to be generalisable through the design process presented: system state, actions, and the reward function, and demonstrated through the design of the contrasting lazy and basic agents.

III **Realistically Implementable**: The system state used by our agents (particularly when using decentralised learning) only has access to local and realistically accessible data, making them more applicable to realistic designs and experiments. Furthermore, the SRL agents offer greatly reduced resource requirements than comparable deep RL options by having a smaller neural network model.

IV **Dealing with catastrophic failure**: Through our experiment with catastrophic failure we showed that our agents can respond to unexpected changes in their environments. Furthermore, the agent's ability to adapt to changing environments (e.g. heterogeneous agents or new types of jobs) improve generalisability through self-adaptability.

This wraps up the evaluation of our learning agent, which has been shown to offer numerous advantages for FPGA-augmented smart IoT devices in complex applications.

7.4. Concluding Thoughts

All three phases of the project have been evaluated here, proving the validity and finding the shortcomings of each.

In this work we have created a fully functional experimentation platform for heterogeneous embedded systems in the Elastic Node. We have also proven that complex and highly capable hardware acceleration can be deployed to it, supporting sophisticated applications such as neural networks and IPCA. Lastly, a number of decentralised agents have been developed that can use RL and AI to learn how to optimise their individual and cooperative behaviour.

In general, our system has been shown to satisfy the requirements set out in Section 2.4.

I **Local Intelligence:** Through presented hardware design optimisations, complex computational problems such as neural networks and IPCA have been shown to be effectively and efficiently computed on the Elastic Node.

II **Automated Cooperation Optimisation:** A sophisticated AI has been presented that optimises device behaviour through cooperation with peers, efficient batching and offloading.

III **Energy Efficiency:** The Elastic Node hardware platform was shown to combine a low power footprint with high computational performance for high energy efficiency.

IV **Convenient and Efficient Local Accelerators:** The deployment of HWFs to the local FPGA was shown to be both fast (switching within 100 ms) and convenient through the provided stub-skeleton abstractions.

What remains is that after discussing some related work next in Chapter 8, we will conclude by revisiting our hypothesis. At this point, we will conclusively prove or disprove it, and highlight some possible future work that could expand on ours.

Chapter 8.

Related Work

The fields of heterogeneous embedded devices, hardware accelerated AI, and AI-based behaviour optimisation have all been very popular in recent years. When setting out to investigate our hypothesis, we analysed the solutions available from literature as well as COTS. The results are presented here, focussing on why the available options did not satisfy our requirements as set out in Section 2.4, or simply how their approaches differ from ours as we developed the Elastic Node.

This chapter expands on the related work presented within the fundamentals in Chapter 3, which covers the core works that form the basis for each topic. In this chapter, we will instead focus on adjacent projects in the literature that attempt to solve similar research questions. The aim is to clarify some of the design decisions made in this work by discussing the results achieved by others.

Consider for example a hybrid IoT solution where most of the heavy computation is offloaded to the cloud, while some basic processing is performed by the local device. The system from Leroux *et al.* [132] trains a full AI model on a GPU-accelerated host and forwards it to the IoT devices (in this case an Nvidia Jetson [173]). Here it can then further be subdivided into smaller networks that can fit into a limited memory footprint. Similarly, Song *et al.* [224] created their so-called in-situ solution for IoT applications, training labels and features using a full cloud network through an unsupervised pre-training method, and then copying parts of it to another inference network.

The fundamental difference between this and the Elastic Node runtime is that in our solution the application complexity is skewed towards the local embedded device [33, 35]. Instead of uploading all captured data for processing and training in the cloud as explained above, we augment the local device with enough application intelligence that raw data does not need to be offloaded. Other approaches that follow this need to simplify the computational problem substantially [24, 81] in order to fit the computational budget of the local device.

We begin in Section 8.1 with an overview of related work in regards to the platform itself. Our Elastic Node runtime includes both a hardware platform and a middleware that supports development, and therefore we discuss here some related work for both. After this, alternative approaches to our learning AI are discussed in Section 8.2. This includes some conceptual work in topics such as Self-Aware distributed systems in Section 8.1.6, as well as various offloading optimisations in Sections 8.2.1 and 8.2.2.

8.1. Platform Related Work

As we set out to perform research in the field of self-adaptive hardware acceleration-augmented embedded devices, we quickly realised that existing platforms did not satisfy our requirements as set out in Section 4.2. Therefore, we decided to design a more appropriate experimentation platform in the *Elastic Node* platform – consisting of both a hardware platform and a software runtime.

Starting with the physical hardware platforms in Section 8.1.1, we also discuss here some available middleware solutions in the form of general heterogeneous systems (Section 8.1.2) and specifically embedded systems (Section 8.1.4). Finally, we will cover briefly some of the available FPGA-based middleware systems in Section 8.1.5.

8.1.1. Heterogeneous Embedded Platforms

Available augmented heterogeneous hardware platforms can be divided into a number of categories. Note that this section does not focus on the software abstractions provided by these platforms (if any) but instead on the hardware itself. Due to the multitude of available projects, we have limited discussion here to hardware that

a. offer low enough power consumption to be considered embedded, and

b. incorporate local on-device hardware acceleration.

This excludes a variety of different hardware platforms, such as the RCMW project [120] which relies on high-end communication interfaces such as PCIe. These are not available on our low-energy MCUs and FPGAs (e.g. Spartan or Artix series from Xilinx). Other projects use fundamentally different configurable logic devices such as a Complex Programmable Logic Device (CPLD) [192] and are therefore not as practical for a wide variety of different applications (breaking our $SREQ_I$: Local Intelligence and making it more difficult to use existing FPGA-focussed designs). Lastly, many projects utilise FPGAs like the Virtex or Zynq SoC series which require multiple Watts of power [90, 130, 155], generally targetting a different application case such as industrial deployments.

Excluding platforms lacking on-device hardware acceleration removes some FPGA-based platforms that emulate multicore sequential processors. For example, the PULP [56] project with its OpenRISC cores and PR-HMPSoC [171] do not offer a substantial performance or computational efficiency increase over traditional MCU-based platforms. All of this led us to create the following categorisation of the available approaches, based on the main components used.

Single device FPGA or SoC systems Among solutions that deploy the application logic exclusively on an FPGA or SoC, various different approaches exist to utilise the available hardware resources. We organise them here based on our own classification, focussing on how usable resources are modelled.

The first variant effectively has no central controlling unit and consists of many simple computational units. These units can be at different levels of complexity, usually representing simplified CPU cores [61]. This type of approach requires a custom compiler [71] that not only creates compatible machine code for the special instruction set used, but also describes how the application is divided amongst the available cores. This often requires developers to write special code to be compatible with this compiler, and require a detailed understanding of the architecture being used to create efficient applications.

The second variant is popular in SoC solutions, and involves deploying a dedicated soft CPU that controls the application logic. Traditionally this utilises hardware-software co-design to develop the platform and accompanying application software, as can be seen in the popular HiReCookie [244] project from Valverde *et al.*. One particularly interesting project is Chimera from Aras *et al.* [7], which utilises a flash-based IGLOO Nano FPGA directly connected to a Radio Frequency (RF) device. This approach requires all the management and flash control logic to be implemented directly on the FPGA.

In some cases, this soft CPU runs a normal OS such as Linux [208] to create a familiar and convenient development environment at the cost of increased power consumption over a bare metal system. One example of this is the Egret project [19, 258] which also allows the addition of hardware modules in a tiled way. Having this multi-layered modular approach increases the computational capabilities of the system – but also increases development complexity when high-level abstractions and code generation is not provided.

Generally, SoC and FPGA-only solutions offer physically smaller hardware solutions due to requiring fewer components, but reduced flexibility for the experimentation and prototyping phases. Instead of being able to exchange individual components to experiment with different platform configurations, an entirely new device needs to be designed. One exception for this is the ability to change the type of soft CPU that is used, but this impacts remaining configurable logic for hardware accelerators.

The last option involves no application logic at all on the FPGA, similar to a PC where the heterogeneous system is composed of different boards connected using standardised ports. This is used for example in the P-HAL project [196], where a somewhat embedded FPGA is abstracted from the system using a PCIe connection. This provides tremendous expandability since CPUs can address multiple PCIe channels, but greatly increases its energy requirements as smaller components (MCUs and FPGAs alike) do not support these types of interfaces.

MCU-FPGA combinations Another alternative is to use separate MCU and FPGA components as is done in the Elastic Node hardware platform. This creates a more adaptable solution where individual components can be exchanged without changing the design of the device, but complicates the communication between the different parts of the application.

One variant of these systems use flash-based FPGAs where the configurable logic is stored in flash memory instead of SRAM (e.g. PowWow [18], HaLoMote [72, 73]). This

means that they do not have to self-configure at boot and therefore start up considerably faster (and without the energy overhead involved in loading their configuration from non-volatile memory). However, they do not have the option of reconfiguring to an arbitrary number of different configurations (even when using dual-configuration devices such as the Intel MAX 10[1]) – making them less applicable to multi-application scenarios (or individual applications that utilise multiple different tasks).

Cookie [123] (the predecessor of the HiReCookie mentioned previously) as developed by Krasteva *et al.* is one of the original MCU-FPGA combinational nodes. It combined a small 8-bit MCU with a Spartan 3 FPGA. It has a very small energy footprint, but offers little in the way of distributed computing capabilities or software abstractions and relies on explicit HW/SW co-design to create custom solutions for every application. Another similar but historically important platform is SENTIOF [217] from Shahzad *et al.* and its descendant SENTIOF-CAM [106]. As SREQ_{IV}: Convenient and Efficient Local Accelerators states, our aim is to make it simple for users of our system to develop their applications. Therefore, we require an abstraction layer for simplifying the interfacing between the main application on the MCU and the accelerator on the FPGA. This calls for a more sophisticated runtime than can be found in the aforementioned platforms, including abstraction layers that hide the complexities of offloading in heterogeneous devices.

8.1.2. General Distributed Heterogeneous Middleware

The field of distributed middleware has been popular for a long period of time (e.g. CORBA [248], DCOM [101], and BASE [15]), but even when aimed at lower powered devices such as mobile phones [210] they cannot be targeted at embedded MCUs and FPGAs. Regardless, much can be learnt from their designs and implementations.

One example of an MCU-FPGA system is CaRDIN [129], which combines Internet Protocol (IP)-based and Virtual Machine (VM)-based middleware onto a combinational ARM+FPGA node. To enable IP-based embedded nodes they rely on Linux as an OS, and extend it with basic web capabilities to provide a RESTful interface directly on the device. Employing regular Distributed Object APIs (DOAs) allows them to collaborate between different nodes through RPCs.

Similarly to Barba *et al.* [13], Villanueva *et al.* [247] uses the ZeroC project to create a distributed object middleware. It references different additions to the embedded OS TinyOS – TinyLime, TinyDB, and Sensation – and addresses everything from tiny 8-bit MCUs to full PCs by generating an ad-hoc message-handling middleware. It creates a static implementation that assumes all objects are on forever, without any explanation of computation support or offloading. This is inopportune for adaptive and dynamic applications where components should be migrated and created/destroyed at runtime.

[1] `https://www.intel.com/content/www/us/en/products/details/fpga/max/10.html` (last visited: 2021-12-13)

Application-specific systems (e.g. [196]) have been developed for simplifying applications such as software radio. It allows sharing data and monitoring statistics between different physical platforms using a bridging concept. It creates a First In First Out (FIFO) interface for packages between different accelerators and the host system. However, little information is provided on the software space or the FPGA-side abstractions.

Similarly, industrial applications commonly utilise a Virtual Bus approach [163, 219] along with a federation approach that interfaces with an Runtime Infrastructure (RTI). This creates a sophisticated system that arguably would not scale to large-scale systems with simple components, due to high complexity on each host.

A fairly manual approach is followed to create a distributed smart camera system by Lewis *et al.* [134], who explains the intricacies of collaboration between smart cameras in detail. The heterogeneity in these systems comes not from hardware variations, but rather behaviour configurations – creating a large solution space when many devices are considered. Emphasis is placed on performance optimisation rather than application development.

8.1.3. Offloading Mechanism

Another area to consider is the mechanism for offloading, which describes how the offloading procedure is performed for moving tasks between nodes. Of particular interest are projects that focus on IoT offloading from embedded devices, which generally focusses on sending tasks to edge/cloud servers. For example, the Hive [220] project is edge-based and implements sound-based emotion recognition. It uses concepts from TinyOS for the embedded OS, and focusses on data-driven collaboration to stream collected data from any device. It uses a simple leader-election system to orchestrate cooperation and organisation.

A survey by Olteanu and Tapus [175] describes many classifications such as partitioning schemes and resource discovery. Discussed systems generally use cloud-style offloading between mobile phones and servers, and includes interesting discussion of why we offload: if it is impossible to fulfil requirements on the mobile client, or simply more convenient to not duplicate the data down to the user.

Dianne [63] presents semi-offloading-based neural network calculation on 'IoT hardware' such as a Raspberry Pi[2]. It splits each layer of the network into its own module that can be individually offloaded, creating concerns for data and communication overhead. Additionally, it has considerable requirements on the MCU, as it uses the high-end Java-based framework OSGi[3]. Skarlat *et al.* [220] brings up offline independence with their REST-based fog computing offloading framework. Their fog orchestration control node can also directly offload to other nodes if required, and is split up into control/reasoning/listening/monitoring/database and other components.

[2] `https://www.raspberrypi.org/` (last visited: 2021-12-13)
[3] `http://www.osgi.org/` (last visited: 0008-05-2022)

Since the offloading mechanism is not the primary focus of this work, a simple implementation was incorporated into the Elastic Node middleware. It allows tasks to be offloaded both locally to the FPGA and remotely to peers using the provided stub, with the device agent making the complex decision of where to compute each task.

8.1.4. Distributed Embedded Middleware

The main objective of most distributed middleware developed for embedded systems is to provide programming abstractions that simplify the development of complex applications spanning multiple devices. This commonly involves the abstractions of some basic OS interactions such as network communications and location services. For example, a traditional approach is followed by Costa *et al.* to create the RUNES middleware [57] using a component model to coordinate these functionalities.

The survey by Razzaque *et al.* describes multiple middleware implementations [194] that follow different approaches – using a variety of different abstractions (e.g. service, event, tuple-based). They also describe some of the functional and non-functional requirements for distributed middleware in the IoT, detailing characteristics of the IoT that impact the design of such systems (e.g. scalability and real-time requirements). Similarly, Palade *et al.* surveys in various works [179, 180] the current trends in IoT-based middleware, with the latter work focussing specifically on Quality of Service (QoS) and Service Level Agreements (SLAs). Required services are largely data-driven, focussing on composing larger services in the system that interact with smaller services to fetch specific pieces of information. It deploys the middleware to VMs in an edge computing fashion, which does not translate well to limited resource embedded devices due to the memory overhead.

Other options offer more application-specific middleware, such as the one addressing UAVs by Barba *et al.* [13]. The focus here is on simplifying development, specifically decoupling hardware and software development. They create an object-orientated middleware that abstracts memory access and hardware/software interfacing through register and interrupt abstractions. Bus access is provided through a Flexible Static Memory Controller, while software is developed within the uC/OS-II OS on an ARM MCU. Internally it relies on the IceC middleware system – which is based on ZeroC. Their results show that their software stack employs a few kilobytes of code (with an overall footprint of roughly 7kB) while the OS is known to require 6kB-24kB. Although relatively small compared to other alternatives from literature, this leaves limited room for other components in smaller flash storage devices (e.g. 32kB) – especially when OTA firmware updating is required.

Some other application-specific Visual Sensor Network (VSN) distributed platforms are described by Rinner and Wolf [197], explaining briefly how Linux-based middleware can be used to do collaborative visual sensing. Little information is available on the details of the distribution of workloads. A similarly specific solution for control systems is available in the work done by Lee *et al.* [130]. It uses the EPICS middleware on a Zynq

FPGA, and employs Linux – again indicating a different resource and power target.

8.1.5. FPGA-Based Middleware

Several frameworks have been developed for server- and PC-grade systems that use FPGAs as coprocessors [60, 109, 120, 170, 182, 222]. Recent research includes FPGA-based coprocessors for the mobile CPUs [77] commonly used in smartphones. These approaches all have a underlying system software or OS that offers a thread-based programming abstraction to interact with the FPGA. OS drivers or library implementations that control the FPGA are hidden behind this threading abstraction (e.g. ReconOS [2]).

Many approaches such as the ARTICo^3 system [200] – from the group that developed HiReCookie – are aimed at the Zynq 7000 series of SoC devices that feature an ARM-based Linux OS. Using this same framework Alcalá performed extensive work on MCU-FPGA collaboration on embedded systems in their PhD [4]. This system combines a MicroBlaze softcore with Dynamic Partial Reconfiguration (DPR) accelerators on a Spartan 6 using the AXI bus, while interactions are abstracted to thread blocks.

The popular LEAP system [182] uses a custom programming language for creating compatible accelerators. It offers a large selection of services for developing sophisticated applications (e.g. memory scratchpads). An RPC interface is created which allows high-level interaction between the applications on the CPU and FPGA, and introduces remote memory for Direct Memory Access (DMA) systems. Although they provide a large set of tools for configuring and compiling applications, it should be noted that considerable boiler plate code is required to create an application using this type of system – increasing the learning curve for new developers.

Other projects provide alternative abstractions such as the work done by Brzoza-Woch and Nawrocki [30] and Min *et al.* [157] where unused FPGA resources are offered via web servers for convenience. Some industrial solutions to heterogeneous middleware uses standardised bus systems such as the Virtual Bus [219]. That allows them to use existing designs that use these interfaces without further development effort, but introduces overhead in order to be compatible. Although these projects are primarily applicable in industrial applications, it is interesting to note that they use the Qsys ARM design tool to map an FPGA to MCU memory space.

In terms of adaptive platforms, Kavanagh *et al.* created a self-adaptive platform for high-level FPGA or GPU abstraction [114]. A sophisticated middlware is used to abstract heterogeneous devices using a number of system components. Although targetting HPC applications and not resource-limited embedded ones, they offer valuable insights into how heteregeneous systems can adapt using variable job execution systems (balancing speed, power, and available resources).

Our approach to making embedded FPGA resources universally available within a distributed network of devices was published as part of the Elastic IoT [33]. Its primary objective is to support developers in using and deploying hardware components on FPGAs within a group of devices by offering a set of tools. This creates a greatly simplified

interface to utilise and control the different aspects of the FPGA, such as data exchange or reconfiguration.

8.1.6. Self-Aware Distributed Systems

The concept of self-awareness relates well to autonomous robotics where each device needs to monitor its own state – mainly battery levels, network bandwidth usage, and activity. One example of this is the work done by Akbar and Lewis, where a Raspberry Pi-based device needs to balance its own power consumption and network usage by gathering knowledge about both itself and its environment [3]. Similarily, Guettatfi *et al.* use environmental and local sensor data for P2P distribution [90]. This results in a Zynq-based VSN that uses an existing middleware (Ella [64]) and OS (ReconOS [2]) to create abstraction layers for computation. Handing off workloads is done using an auction-based system that creates a graph of nearby and related nodes using its self-awareness. It also considers system performance of accelerators and buffer loads to balance multi-core utilisation.

A similar approach for multi-core systems was used by Happe *et al.* [93] by the group that created HiReCookie and ARTICo^3, including a study of power usage for different sized accelerators. The work was extended [93] to include a comparison of the hardware/software threads in a video object tracking application. They also describe their self-aware component [199], detailing resource usage during the different stages of their thread block: reconfiguration, data transfer, and kernel execution.

8.1.7. System Modelling for Distributed Systems

Nam and Lysecky created a latency and energy-based modelling for optimising an embedded distributed system [169]. Furthermore, the data-flow of the application itself is modelled to optimise its performance. They follow a placement approach that tries to distribute a specified application across a given set of devices. A data-flow model specifies how much data needs to be passed between each set of tasks, while execution and communication latency models use measurements for each option. This considers hardware/software execution time, and handing tokens between components on one or on different devices. The design space is optimised using a generic algorithm that ensures constraints are met and then optimises for overall power usage. Their solution is centralised, and appears to ignore the reconfiguration overhead of both hardware and software.

The ARTICo^3 system detailed by Rodriguez *et al.* [199] includes a detailed power monitoring of memory and computation-based power usage. This leads to a simplified model that estimates power usage when using any number of accelerators. This model is empirically created during development and evaluated using hardware. However, their system does not address effects of distributed systems on these models.

8.1.8. self-x Reconfigurable Systems

One approach to incorporating self-x systems into reconfigurable systems is by allowing them to adapt themselves. Although computing architecture designs are normally very rigid and fixed, techniques such as Error Correction Code (ECC) memory (where the memory can detect and correct small errors due to data corruption) can improve dependability and reliability.

Under the umbrella term of *dependable embedded systems* a number of approaches have been developed that utilise reconfigurable computing to compensate for errors due to external factors (e.g. heat or ageing) or internal factors (e.g. design errors) [96]. One particularly relevant example is the SMASH project that combines hardware threads (see Section 3.3.1) and tile-based architecture design (see Section 3.3.2). Error-prone threads due to thermal hotspots are managed by changing the deployment locations of each piece of computation.

Somewhat similarly, autonomic SoC designs can adapt to failing processing cores by utilising a learning component (based on an LCS [21]) and an organic computing middleware [25]. A multi-core system was shown to adapt to changing environments and workloads by learning a set of adaptation rules (e.g increasing clock speeds to accommodate high workloads) in its classifier system [279].

8.2. Intelligent Embedded Offloading

Device intelligence can take many forms based on how the *intelligence* is defined. We focus here on the different ways for devices to *learn* and *understand* how they should act. This includes high-level concepts such as self-awareness, as well as explicit and low-level approaches that rely on detailed modelling and rules. The objective here is to classify different approaches to introducing intelligence into deployed embedded systems – a popular objective in related work.

A number of approaches can be found in literature that attempt to introduce intelligence to offloading embedded systems. These vary from direct approaches that attempt to accurately predict the energy costs of various offloading options, to more conceptual ones that aim to achieve higher level goals. A quick overview of a selection of these approaches can be seen in Table 8.1, where we provide a breakdown of how each approach models the system state, the action space, and the cost function. This makes it clear that some metrics are shared amongst almost all of them (e.g. task delay), while others are more rare (e.g. device power state).

A characterisation of these (and other) approaches is provided here – grouped by the type of approach. The research question they have in common is how to solve computational problems created at various embedded devices. Apart from the simplified scenario where each device can realistically solve their problems locally, this generally involves offloading to the edge or a server. Alternatively, P2P solutions also consider distributing these computational tasks directly between the devices – offering independence from the

outside world.

The fundamental issue defined in all these works is the binary offloading problem: should the device compute the task locally or offload it somewhere else. A notable extension to this is the partial computation offloading problem, where tasks can be broken down into smaller segments and offloaded independently [151]. Another variation is the mixed-binary extension by Xu *et al.* who considers a choice of different offloading servers [271]. Alternatively, a multi-layered offloading problem is created by Chen *et al.* [40] – but it appears that from the perspective of the local device the problem remains binary.

8.2.1. Greedy and Short-Term Learning Approaches

Some approaches are directly concerned with predicting the expected costs of various options – e.g. comparing offloading with local computation. As they generally assume that locally modelling these energy costs are not feasible due to lack of reliable data, most of them use some estimation algorithm to predict costs instead.

Lyapunov One segment of these utilise Lyapunov optimisations [41, 138, 150], which was originally developed to solve the equilibrium point of Ordinary Differential Equations (ODEs). These aim to optimise a requirement system-wide, which makes them ill-suited to our scenario which relies on a decentralised solution as part of IREQ_{III}: Realistic Implementation.

MDP Alternatively, other projects have utilised MDPs [231, 280]. Here Tang *et al.* [231] builds on the work done by Zhang *et al.* [280] by creating a decentralised system that can function on a partially observable local environment. Similar to the MEC and MCC devices, the fog IoT devices considered are simple CPU-based embedded devices based on EH.

Some approaches also limit the task queue size to minimise the exponential state expansion [280], which is also the case for graph based modelling approaches [91]. Here the size of the state space directly impacts the feasibility of the search space. A similar issue exists for the projects that utilise game theory. Ma *et al.* [144] and Chen [44] formalised the competing goals of the users and devices, thereby creating a decentralised game that optimises computational offloading.

SDR Semidefinite Relaxation (SDR) was utilised by Dinh *et al.* [65] to simultaneously optimise task offloading and local frequency scaling. This enables them to optimise offloading to multiple access points, but the complexity of their centralised solution limits its applicability to device-local distributed solutions.

Game theory Some approaches use game theory [44, 136] to formalise the competing goals of different devices, creating a so-called decentralised computation offloading game.

Fundamentally, game theory requires each agent to create a policy by creating a payoff function for each combination of its own possible actions and that of other agents [206]. This allows it to create a strategy that allows it to maximise its own benefit based on some understanding or assumption of the strategies followed by other agents. Alternatively, they may communicate to try reach some kind of consensus for mutual gain.

8.2.2. Reinforcement and Deep Learning approaches

Using RL and DL approaches have gained recent popularity in the MEC and MCC communities. In general they directly address the offloading problem, creating an agent that decides whether incoming work should be locally computed or offloaded to a server. Xu *et al.* [271] and Huang *et al.* [104, 105] both created a cloud-based deep RL agent that solve a multi-device offloading problem in a centralised fashion. As is common in similar approaches in the MEC, MCC and Radio Access Network (RAN) fields, communication is emphasised while the computational tasks themselves are simplified – in this case modelled as offloading data streams.

Neural network-based solutions fall into two different camps: one where the 'deep' nature of the network is questionable, and one where the networks are so large that one is in danger of overfitting. Others simply do not offer details on their networks [272], making it difficult to know if their solutions would generalise. Some examples of small networks are from Li *et al.* [135], Liu *et al.* [139] and Chen *et al.* [43], both of which use DQN to optimise a cost function for balancing server load with task deadlines. Additionally, DQN overcomes catastrophic forgetfulness [183] by retraining on old data in order to avoid erasing weights of previously learned state-action pairs.

In contrast to this, Salmani *et al.* [207] created a network consisting of a total of over 24,000 nodes (8 hidden layers with 3000 nodes per layer). At this point, the ability of the network to generalise to other problems or changes in the scenario comes into question, which stands directly against IREQ_{II}: Generalisability. Although it outperformed other methods in CPU time, one could argue that its performance would decrease if presented with more than two candidates. In other words, it would not be able to respond to dynamic changes in requirements. A similarly large network is used by Clausen *et al.* [54] in their case study of the game 'Connect 4'.

This is a major consideration in the design of our SRL agents. By using a much shallower neural network, the system can better relearn behaviour as required. Although not a concern in static environments, our aim for the learning device agents is to respond well to changes as stated in IREQ_{IV}: Catastrophic Failure.

This is backed up in the work done by McDonnell *et al.* [154], where they show that their Extreme Learning Machine (ELM) approach can perform just as well as deeper networks on the Modified National Institute of Standards and Technology (MNIST) classification test sets. Similarly, Jiang and Crookes have shown that their Small Unorganized Neural Networkss (SUNNs) perform well on early image recognition tasks [111]. By combining adaptive neurons and random interconnections in a 2D structure, their

network can learn early edge detection faster than state-of-the-art DNNs. Their experiments show that in less complex applications their SNNs are at an advantage to larger, deeper networks.

The distinctions between the Elastic Node runtime and alternatives from literature have been described here. This included details in the design of both the hardware platform itself and the middleware providing development support. Similarly, the intelligence it provides by locally optimising offloading between devices provides a novel approach for a learning embedded system. However, it is essential to consider the Elastic Node's place within similar projects both from the world of academia and commercial industry. This provides important context for what we have accomplished in this thesis.

	Xu *et al.* [271] (RAN)	Chen [44] (MCC)	Chen *et al.* [42, 43] (MEC)	Huang *et al.* [105] (MEC)	Li *et al.* [135] (MEC)	Mao *et al.* [138, 150] (MEC)	Zhang *et al.* [280] (MCC)	Guo *et al.* [91] (MCC)	Chen *et al.* [40] (MCC)	Ma *et al.* [144] (MCC)
System State										
Power state	x									
User demand	x								x	
Offloading decisions		x	x	x				x		
Interference		x	x	x		x		x		x
Energy queue			x			x				
Harvestable energy						x				
Task arrival/queue			x			x	x			
Total system cost					x			x	x	x
Available server resources					x		x			
Application phase							x			
Actions										
Power state control	x					x(f)		x(f)		
Server selection	x								x	x
Binary offloading decision		x	x	x	x	x	x	x	x	x
Wireless allocation				x				x		
Power transmission				x		x				
Server allocation					x				x	x
Cost										
Power consumption	x	x			x	x		x	x	x
Task delay		x	x	x	x		x	x	x	x
Task dropping cost			x			x	x			
Server cost			x						x	

Table 8.1.: Some projects from related work from mainly MCC and MEC. Includes system state collection, actions, and decision cost computation, highlighting how each application has its own states it considers important to the device/system. Some actions are very rare (e.g. altering the device frequency indicated by (f)) while others are almost universal (e.g. the binary offloading decision).

Chapter 9.

Conclusion and Outlook

Augmenting embedded devices with local FPGAs has been shown to greatly increase their processing capabilities. In a modern world where some form of AI is ubiquitous to almost every system, they create more capable and flexible devices that can perform complex applications without depending on the cloud. To address the difficulties of optimising a distributed system of such devices, we created a device AI that allows it to learn how to behave rationally.

Our original hypothesis from Section 1.1 was that

Connected and autonomic FPGA-augmented smart IoT devices can use AI to optimise their behaviour.

We identified a three-phase approach to answer this hypothesis, dedicating a chapter to each phase. Firstly, the Elastic Node platform was developed in Chapter 4. This involved creating both an appropriate hardware and software environment for autonomic FPGA-augmented smart IoT devices. Secondly, the capabilities of the local AI were maximised through a number of known hardware optimisation techniques in Chapter 5. This included a number of case studies of popular AI and ML techniques (specifically ANN, CNN, and IPCA). Finally, Chapter 6 developed a novel behaviour optimisation using RL and Q-learning. We demonstrate its ability to govern a team of Elastic Nodes first using a basic Q-table, as well as a neural network for improved resilience to changes in the system state and environment.

9.1. Contributions

The main contributions of this work (as enumerated in Section 1.3) were addressed through these three phases (with Phase 1 covering the first two contributions). All of the contributions were evaluated in Chapter 7, and various parts were demonstrated in person to the scientific community.

In Chapter 4 we introduced the Elastic Node as *a novel hardware platform for FPGA-augmented connected IoT devices* (Contribution 1), accompanied by our middleware which reduces *the development complexity of embedded heterogeneous applications* (Contribution 2). The hardware was evaluated for its viability as an experimentation platform, showing that it has very low power consumption and offers detailed self-contained

per-component energy monitoring. Within a power envelope of 200 mW, the Elastic Node platform supports multi-accelerator FPGA-MCU applications that can change active HWFs within 100 ms. This was demonstrated at PerCom 2018 [31] and ICAC 2019 [211], where the Elastic Node's ability to outperform offloading and provide meaningful experimental results were illustrated to the community.

Without introducing substantial FPGA resource overheads like other solutions from literature, our platform lowers the development complexity by offering RPC-like task offloading to the local FPGA. Application developers can utilise the generated stubs to add on-device intelligence by deploying the relevant HWF. Similarly, accelerator developers can use our IDL to describe any compatible HWF's interface – after which the required stub and skeleton is generated. Along with the provided Hardware Abstraction Layers (HALs), this formed part of a journal article in the Future Generation Computer Systems journal [33].

Next, Chapter 5.1 addresses *creating hardware accelerators that efficiently utilise the available resources* (Contribution 3). Although the small embedded FPGAs supported by the Elastic Node offer very limited PL resources, we discussed various ways to utilise them optimally when adding local intelligence to smart IoT devices. By employing a number of hardware accelerator design techniques, we developed local AI accelerators previously impossible on this class of device. Specifically our IPCA accelerator as published at ARCS 2020 [36] is considerably more resource-efficient than what is available in the literature.

In our evaluation we found that for larger covariance matrices our design used 84% fewer DSP slices than CORDIC-based alternatives. We also identified various ways of parametrising designs to balance resource consumption and processing speed/accuracy requirements. For example, the various impacts of altering the fixed point representation for our CNN implementation were analysed, focussing on power consumption and resource usage. Additionally, this parametrisation allowed us to develop latency models that can accurately predict each HWF's performance. Each of our developed HWFs were presented in conference and workshop papers (e.g. the CNN at the PerIoT workshop at PerCom 2020 [35]).

The development of a learning algorithm for controlling the behaviour of teams of Elastic Nodes forms the cornerstone of Chapter 6. By using Q-learning and RL, we showed how devices can learn how to behave rationally [206]. This was done by *extending on the offloading problem to fully utilise FPGA-based hardware acceleration* (Contribution 4), thereby allowing them to collectively perform 150% more jobs in the same power budget than a randomly-acting agent. The local AI component formed the basis of our award-winning paper at ACSOS 2020 [34], and was extended with the SRL agent in the ACM Transactions on Autonomous and Adaptive Systems [37].

Through our evaluation in Chapter 7 and the mentioned publications/demonstrations, the validity of the Elastic Node platform have been shown. Between our hardware/middleware design, hardware accelerator designs, and learning agent designs, a variety of contributions have been made to the respective scientific communities.

We envision that our work will enable a wider audience to get involved in developing smart IoT devices. Using our platform as foundation, users can focus on their own objectives — whether that is creating experiments with FPGA-augmented embedded devices or developing powerful self-aware device agents. By assisting developers and putting a full system solution (hardware platform, software runtime, and local device agent) into their hands, we hope to collectively advance the state of the art.

9.2. Research Questions

Our objective was to address two research questions as detailed in Section 1.2:

- **RQ1:** How can adaptive hardware acceleration increase the local intelligence of IoT devices?
- **RQ2:** How can distributed heterogeneous embedded devices learn to autonomously achieve shared goals?

We identified the main way to increase local intelligence of IoT devices in Chapter 5 to be supporting and optimising the hardware architecture design. This allowed us to utilise considerably more complex and powerful ML and AI algorithms than is normally possible using basic MCU-based designs. This is further improved upon by using good hardware accelerator design techniques to further increase local AI capabilities.

Similarly, our work in self-governing intelligent devices detailed in Chapter 6 illustrated how our distributed heterogeneous embedded devices can cooperate autonomously. By introducing intelligent behaviour through on-device RL, we showed how teams of Elastic Nodes can learn to achieve shared goals. By optimising the augmented offloading problem (considering local FPGA task acceleration and peer-to-peer offloading), these devices achieve considerably better application performance.

Through the contributions of all three phases of this work, we have thoroughly answered the hypothesis stated in Section 1.1. By maximising the local AI available on a combined MCU-FPGA embedded platform, it can learn to optimise its behaviour through RL and Q-learning. This allows a set of developers to create a smart IoT device capable of dynamically adapting to its environment by cooperating with nearby neighbours.

9.3. Outlook

The current state of the art in the field of heterogeneous embedded devices is very exciting. Academic works such as this one have proven that complex and sophisticated self-optimising systems can be created that utilise flexible local hardware acceleration. At the same time, the industry is creating COTS devices that provide ever-growing

performance capabilities at increasingly better energy efficiency. However, great opportunities still exist to expand the capabilities of these devices.

For the hardware platform itself, further development can always improve the overall performance or energy efficiency. To explore additional experimentation options, the usage of other processing architectures such as 32-bit ARM or Reduced Instruction Set Computer (RISC) cores could create new possibilities for MCU-dependent applications. Similarly, while we focussed on very resource-limited FPGAs in our platform, larger and more powerful FPGAs such as the Kintex range from Xilinx would expand its applicability to a new set of applications. Instead of being limited by the power supplied by a small battery, such a platform could address other fields such as the automotive industry. Using our design to optimise the software and FPGA-resource overhead, we envision this larger version of the platform to perform advanced real-time AI such as computer vision used for autonomous driving [223].

One could expand the management of multi-phase accelerators or more complex task graphs that require multiple accelerators in succession [52]. This would offer a different option for creating more advanced applications without necessarily using larger components. To further improve the cooperation of multiple devices in a system, a system-wide approach that improves the management of the available computing resources would offer new opportunities [33, 53]. Instead of relying exclusively on the local device intelligence to learn a more optimised solution, a user-in-the-loop approach can provide more contextual improvements that leverage domain knowledge.

Additionally, the middleware software system in our runtime could be expanded from the purely bare metal system in the current Elastic Node runtime to use an RTOS such as FreeRTOS [5] or ReconOS [142, 143]. Since our middleware is designed to operate in an OS-agnostic way, it should be straight-forward to port to these software systems, but this could make it easier for developers familiar with those OSs to migrate to our platform. Integration with a full embedded OS would not only expand our platform's opportunities for adoption, but could expand support for more complex applications. Along with utilising more than one hardware accelerator at a time, it would offer a more appropriate way to develop applications that deploy both software and hardware tasks. Wrapping our stub-skeleton abstraction in a task or thread would make it easier to create large applications that rely on multi-tasking.

While the possibilities are practically endless for developing additional optimised hardware accelerators, particularly our work in parametrised accelerators offers great potential for further work. In the same vein as our dynamic CNN and IPCA that can be tailored to different accuracy and resource requirements, other accelerators would benefit from parametrisation. For example, various other neural networks (e.g. graph neural networks [261]) could be adapted in a similar way.

One augmentation that could be made to the self-organising and self-optimising agents created in Chapter 6 is to add further control options and a more detailed system state. The quality of a device agent's control is bound by both its action and system spaces, as these limit its ability to regulate and understand its own state respectively. For example,

additional actions such as controlling the FPGA power state or offloading groups of tasks could be introduced. Additionally, a more detailed view of the system state could be achieved by adding additional sensors to the board such as an RTC in order to learn a time-varied policy that considers what time of day it is.

Another area that offers potential for further work is utilising more advanced training paradigms such as a genetic algorithm [113]. Instead of training a number of independent agents and choosing the top results at the end of the experiment, these techniques provide an opportunity to optimally combine the knowledge of agents between generations. Another opportunity for future work is to improve the creation of high-level objectives for the device. Instead of manually creating a reward function that embodies the user's intent, alternatives such as game theory could offer a more intuitive design tool.

We believe that substantial further developments will be made in the field of smart heterogenous IoT devices, both in this research group as well as the community in general. As applications and requirements are always expanding and becoming more complex, the future looks very bright for more capable and efficient devices to address them.

Working with FPGA-augmented smart IoT devices that optimise their own behaviour provided an opportunity to combine a number of exciting and fascinating fields of research. Fundamentally, we have shown that it is possible to use elastic and decentralised computing to address concerns about privacy and our increased dependency on the cloud. As long as the right assistance is provided to developers, and devices are embedded with enough local intelligence, we foresee a bright future for the IoT.

Appendix A.

Interface Description Language Specification

The specification of the IDL used for the Elastic Node platform is provided here as adjusted from work done by Winnekens [260]. It starts with the grammar itself in Appendix A.1 which describes the construction of writing the IDL in EBNF. This is followed by a description of the available fields in Appendix A.2, the way a HWF (configuration) is mapped in Appendix A.3, and the available variable types in Appendix A.4.

A.1. Grammar

The grammar for our IDL has been broken up into the generic part in Listing A.1, and the sections for describing the connections to the MCU (Listing A.2) and the HWF (Listing A.3). That allows the user to define the full connectivity of the platform through this single IDL, as it allows the stub and skeleton generators to create the required code for the MCU and FPGA respectively.

Code Listing A.1.: Initial section of EBNF for the Elastic Node Platform IDL

```
1 overallConfig       = 'configuration', configName, colon, NEWLINE,
  ↪  mcu, NEWLINE, function ;
2 colon               = ':' ;
3 configName          = STRING ;
4 mcu                 = 'mcu', colon, NEWLINE, mcuConfigs ;
5 function            = 'function', functionName, colon, NEWLINE,
  ↪  functionConfigs ;
6 functionName        = STRING ;
```

Code Listing A.2.: MCU section of EBNF for the Elastic Node Platform IDL

```
mcuConfigs          = wordsizeConfig, addressWidthConfig,
    endianConfig, stateConfig, includeConfig ;

equalsOp            = '=' ;
wordsizeConfig      = 'wordsize', equalsOp, ("1" | "2" | "4" | "8" |
    "16" | "..." ), NEWLINE ;
addressWidthConfig  = 'addresswidth', equalsOp, integer, NEWLINE ;
endianConfig        = 'endianness', equalsOp, ('little' | 'big'),
    NEWLINE ;
stateConfig         = 'activestate', equalsOp, ('high' | 'low'),
    NEWLINE ;

includeConfig       = { singleIncludeConfig } ;
singleIncludeConfig = {'include', equalsOp, includeFile, NEWLINE} ;
includeFile         = (doubleQuotedHeader | bracketedHeader) ;
doubleQuotedHeader  = '"', headerfile, '"' ;
bracketedHeader     = '<', headerfile, '>' ;
headerfile          = STRING, ".h" ;

hdlConfig           = 'hdl', equalsOp, ('vhdl' | 'verilog') ;
typeConfig          = 'type', equalsOp, ('oneshot' | 'asynchronous' |
    'streaming') ;
libraryConfig       = 'library', equalsOp, STRING ;
implConfig          = 'implementation', equalsOp, STRING ;
idConfig            = 'id', equalsOp, byteInteger ;
byteInteger         = "1" | "2" | "..." | "254" | "255" ;
```

Code Listing A.3.: Function section of EBNF for the Elastic Node Platform IDL

```
functionConfigs      = hdlConfig, typeConfig, libraryConfig,
↪  implConfig, idConfig, { singleFunction } ;
singleFunction       =  | mappings ;

hdlConfig            = 'hdl', equalsOp, ('vhdl' | 'verilog'), NEWLINE ;
typeConfig           = 'type', equalsOp, ('oneshot' | 'asynchronous' |
↪  'streaming'), NEWLINE ;
libraryConfig        = 'library', equalsOp, STRING, NEWLINE ;
implConfig           = 'implementation', equalsOp, STRING, NEWLINE ;
idConfig             = 'id', equalsOp, byteInteger, NEWLINE ;
includeFile          = (doubleQuotedHeader | bracketedHeader) ;
doubleQuotedHeader   = '"', headerFile, '"' ;
bracketedHeader      = '<', headerFile, '>' ;
headerFile           = STRING, ".h" ;

mappings             = { portMap | NEWLINE } ;
portMap              = [dataType], targetConn, connectionOp, portName,
↪  NEWLINE ;
connectionOp         = '->' ;
portName             = 'clock' | 'reset' | 'start' | 'done' | 'data_in'
↪  | 'data_out' | 'ctrl_in' | 'ctrl_out' | 'data_in_valid' |
↪  'data_in_ready' | 'data_out_valid' | 'data_out_ready' | 'suspend'
↪  ;
targetConn           = [connectionType], (combinedTarget |
↪  singleTarget) ;
combinedTarget       = "(", singleTarget, { ",", singleTarget, ")" };
singleTarget         = [targetType], STRING ;
targetType           = dataType, [arraySize];

connectionType       = 'pulse' | ('hold', colon, portName) | buffer;
dataType             = 'bit' | 'u8' | 'u16' | 'u32' | 'u64' | 'i8' |
↪  'i16' | 'i32' | 'i64';
buffer               = 'buffer', arraySize;
arraySize            = "[", bitNumber, { ',', bitNumber } "]";
bitNumber            = integer ;
genericTarget        = combinedTarget | singleTarget;
```

For clarity, all of these fields are described in the coming section as well, declaring their purpose and valid values. Additionally, it describes which fields are optional, and how many times each field can be used (once, multiple times, or as many as required).

A.2. Fields

The various fields that can be configured are detailed here. Along with the mapping of ports being connected, this pertains to the main chunk of the IDL's usage.

A.2.1. MCU Configuration

The MCU configuration parameters occur inside the `mcu` block, indicating that they primarily affect the way the MCU accesses or addresses the data. This makes them dependent on the architecture of the MCU, meaning they need to be updated whenever a different MCU is being used. Shown in Table A.1 are the available values that can be configured, some of which are required and others that can be set multiple times as described in Section A.1.

Keyword	Function	Valid Values	Default
`wordsize`	Specifies the width of the used architecture's memory cell	8, 16, 32, ...	`8`
`addresswidth`	Number of bits used to transfer the address in the interconnect	natural number	`16`
`endianness`	Specifies how the architecture stores data in memory, specifically how multi-byte data types are organised	little or big	`little`
`activestate`	General active state of the MCU logic lines	high or low	`high`
`include`	Specifies C header files that should be included when generating the stub. May provide structure/data type definitions.	"string.h" or <string.h>	*none*

Table A.1.: MCU Configuration values in Elastic Node IDL

A.2.2. Function Configuration

The function configuration section details mostly to the skeleton (i.e. the VHDL section of the interface). Therefore, it covers partially accelerator-dependent values (e.g. the type of skeleton required – see Section 4.4.2). These need to be updated once per accelerator only, while other values such as `id` are application-dependent (as it depends on how configurations are stored on the local flash storage). Table A.2 provides an overview of the possible configuration values that specify how the VHDL skeleton should be generated.

Keyword	Function	Valid Values	Default
`hdl`	Language targeted for generated skeleton.	VHDL or Verilog	`VHDL`
`type`	Specifies the type of skeleton to generate based on the interaction scheme (see Section 4.4.2)	oneshot, asynchronous or streaming	*Required*
`library`	Controls which library the VHDL accelerator is compiled into (and therefore how to import it into the skeleton)	string	`work`
`implementation`	Name of the VHDL architecture or implementation for the skeleton.	string	`Behavioral`
`id`	Skeleton ID for uniquely identifying corresponding stubs and skeletons in the middleware.	$0 < id < 256$	*random*

Table A.2.: VHDL Configuration values in Elastic Node IDL

Mapping	Max	Oneshot	Asynchronous	Streaming
`clock`	1	✓	✓	✓
`reset`	1	✓	✓	✓
`start`	1	✓	✓	×
`done`	1	✓	✓	×
`data_in`	m	✓	✓	✓
`data_out`	n	✓	✓	✓
`ctrl_in`	∞	✓	✓	✓
`ctrl_out`	∞	✓	✓	✓
`data_in_valid`	m	×	×	✓
`data_in_ready`	n	×	×	✓
`data_out_valid`	n	×	✓	✓
`data_out_ready`	n	×	✓	✓
`suspend`	1	×	✓	×

Table A.3.: Overview of connections that can be mapped and their corresponding supported skeleton types. Includes the maximum number of appearances for a HWF with m inputs and n data outputs.

A.3. Configuration Mapping

A number of different control and data interface values can be mapped in the skeleton, which is why we provide various known mappings. These differ for each different skeleton type (as described in Section 4.4.2). Therefore, an overview is provided here for the different options that can be mapped, including which skeletons they refer to and how many of each mappings can occur.

In this overview is an HWF with m data inputs and n data outputs. For the *asynchronous* skeleton, the output data is buffered and therefore include `ready` and `valid` lines – these indicate when the skeleton is ready to receive new data and a line specifying when the data is valid and can be read. When using a *streaming* interface, these lines are also available for input data.

Control lines can be mapped to as many `ctrl` inputs and outputs. This provides the ability to set configuration values (and read them out) similarly to how configuration registers are set in most MCU architectures such as AVR and ARM.

A.4. Types

The definition of data types requires separate entries for usage in our Elastic Node IDL, the generated skeleton in VHDL, and in C within the created stubs. A number of typical data types are provided here, covering most of the commonly used in each language respectively. Note that custom data types can be added fairly trivially, or simply converted to an array of specific width as shown in Table A.4. Static types are required, however, for buffering and memory allocation in VHDL.

IDL-Type	VHDL-Type	C-Type
bit	std_logic	uint8_t
u8	unsigned(7 downto 0)	uint8_t
u16	unsigned(15 downto 0)	uint16_t
u32	unsigned(31 downto 0)	uint32_t
u64	unsigned(63 downto 0)	uint64_t
i8	signed(7 downto 0)	int8_t
i16	signed(15 downto 0)	int16_t
i32	signed(31 downto 0)	int32_t
i64	signed(63 downto 0)	int64_t
bit[12]	std_logic_vector(11 downto 0)	uint16_t
bit[32]	std_logic_vector(31 downto 0)	uint32_t
bit[128]	std_logic_vector(127 downto 0)	uint8_t*
u8[8]	array (7 downto 0) of unsigned(7 downto 0)	uint8_t*
i16[4]	array (15 downto 0) of unsigned(7 downto 0)	int8_t*
u8[2,4]	array (1 downto 0, 3 downto 0) of unsigned(7 downto 0)	uint8_t*

Table A.4.: Various data types defined for use in the Elastic Node IDL and their corresponding types in VHDL and C

Using the IDL described here, a user can describe not only one or more HWFs that should be used on the Elastic Node platform, but also how these interface with the MCU. By describing how the system components on the MCU and FPGA communicate, the stub and skeleton can be generated as discussed in Section 4.4. At this point, all that remains is to create the top-level application logic on the MCU using the stubs and other middleware functions provided by the Elastic Node runtime.

A.5. Defaults

To illustrate the usage of the default values, an absolute minimal IDL is shown in Listing A.4. All of the optional values are ignored, and accordingly a HWF is created that takes no inputs and no outputs.

Code Listing A.4.: Minimal IDL description using all the default values

```
configuration Minimal_example:
   mcu:
      wordsize = 8
      addresswidth = 16
      endianness = little
      activestate = high

   function Minimal:
      hdl = vhdl
      type = oneshot
      library = work
      implementation = Behavioral
      id = 1
```

Next, a simple example is provided in Listing A.5. It shows an HWF that directly calculates the result of $A = B + C$, where each variable is a 16-bit unsigned number. The computation is triggered using the oneshot type, which uses the `trigger` line, and uses `done` to know when the computation is done.

Code Listing A.5.: Simple example IDL of a mathematical function calculating $A = B + C$ in a one-shot skeleton style

```
configuration Math_example:
   mcu:
      wordsize = 8
      addresswidth = 8
      endianness = little
      activestate = high

      include = <math.h>

   function Math:
      hdl = vhdl
      endianness = little
```

```
11        activestate = high
12        type = oneshot
13        library = work
14        implementation = Behavioral
15        id = 2

16        bit clock -> clock
17        bit reset -> reset
18        bit trigger -> start

19        buffer[10] u8 A -> data_in
20        buffer[10] u8 B -> data_in
21        bit done -> done
22        buffer[10] u8 C -> data_out
```

Lastly, an example of a streaming HWF is shown in Listing A.6, showing the HWF for calculating the dot product of two vectors. Specifically $\vec{A} \cdot \vec{B} = C$, requiring a streaming input where the two vectors are provided in parallel and the cumulative dot product can be read at any point. Note that the I/O caching depth n is not provided in the IDL, as this is only required by the stub/skeleton at runtime – as this process takes place in software on the MCU. It uses the `ready` and `valid` lines to provide new data to the HWF and fetch the next result. This IDL also takes advantage of the `buffer` functionality to cache 10 sets of I/O on the FPGA.

Code Listing A.6.: Streaming IDL of a dotproduct computing HWF, accepting two vectors to create a single value representing the collected dotproduct

```
1  configuration DotProduct_example:
2     mcu:
3        wordsize = 8
4        addresswidth = 16
5        endianness = little
6        activestate = high

7        include = <math.h>
8        include = "vector.h"

9     function DotProduct:
10       hdl = vhdl
11       endianness = little
12       activestate = low
```

```
13        type = streaming
14        library = work
15        implementation = Behavioral
16        id = 3
17
18        bit clk -> clock
19        bit reset -> reset

20        i16 A -> data_in
21        i16 B -> data_in
22        bit inputRdy -> data_in_ready
23        bit inputValid -> data_in_valid

24        i16 C -> data_out
25        bit CRdy -> data_out_valid
26        bit CValid -> data_out_valid
```

A further example is provided in the Evaluations chapter in Listings 7.1, showing the IDL representation for our ANN HWF. It uses a very similar structure to our simple mathematical example above in Listing A.6.

Appendix B.

ANN Interfaces

As an appendage to the qualitative evaluation of the Elastic Node's development complexity in Section 7.1.4, we provide here the full interface descriptions for both the user-provided VHDL file and the generated stub and skeleton.

As a reminder, the procedure for integrating a new HWF into the Elastic Node platform is as follows:

1. Provide a compatible HWF VHDL entity
2. Identify the appropriate skeleton type using Section 4.4.2
3. Create the IDL representation using Appendix A
4. Generate the appropriate stub and skeleton

This process is illustrated here through the relevant parts for incorporating the ANN HWF. Note that this is provided only for insight into the stub and skeleton generation, as these files do not need to be directly used by the user. Instead, they can simply be included in their respective projects and called using the provided API.

B.1. ANN VHDL Entity

Provided in Listing B.1 is the VHDL entity for the ANN HWF. A VHDL entity describes primarily the port interface of a hardware architecture, and acts as the top level API for incorporating it into a project.

Code Listing B.1.: ANN Entity Description in VHDL

```
ENTITY SignedANN IS
   PORT (
      -- control interface
      clk             : IN STD_LOGIC;
      reset           : IN STD_LOGIC;

      -- data handshake
      data_rdy        : OUT STD_LOGIC := '0';
      start           : IN STD_LOGIC;

      -- data interface
      connections_in  : IN int8_t;
      connections_out : OUT int8_t := (OTHERS => '0');
      wanted          : IN int8_t;

      -- hwf control interface
      learn           : IN STD_LOGIC;
   );
END SignedANN;
```

This interface is very simple but effective. It allows a user to provide the required data inputs (`connectionsIn` and `wanted`), as well as the signal to start a computation (`start`). Once the computation is done, `dataRdy` goes high and the results can be read from the `connectionsOut` output.

This is reflected in the IDL provided in Listing 7.1, where the full description that needs to be provided by the user is shown.

Appendix C.

ANN Skeleton

Once the entity description and the IDL are available, the stub and skeleton can be generated. The result of generating the skeleton is provided in Listing C.1, showing the complexity required for adapting the simple entity shown in Listing B.1 so it can be compatible with the Elastic Node middleware.

Code Listing C.1.: Generated skeleton for the ANN HWF

```
1  -- Generated by fpgasg 0.3
2  -- Date: 2021-01-06 11:42:37

3  LIBRARY ieee;
4  USE ieee.std_logic_1164.ALL;
5  USE ieee.numeric_std.ALL;
6  LIBRARY fpgamiddlewarelibs;
7  USE fpgamiddlewarelibs.UserLogicInterface.ALL;
8  LIBRARY work;

9  ENTITY NeuralNetwork_skeleton IS
10    PORT (
11       skeleton_clock    : IN STD_LOGIC;
12       skeleton_reset    : IN STD_LOGIC;
13       skeleton_write    : IN STD_LOGIC;
14       skeleton_read     : IN STD_LOGIC;
15       skeleton_address  : IN uint16_t;
16       skeleton_data_in  : IN uint8_t;
17       skeleton_data_out : OUT uint8_t
18    );
19 END NeuralNetwork_skeleton;

20 ARCHITECTURE Behavioral OF NeuralNetwork_skeleton IS

21    SIGNAL hwf_dataRdy_s         : STD_LOGIC;
22    SIGNAL hwf_connectionsOut0_s : uint8_t;
```

```
23    SIGNAL skeleton_write_once_s : STD_LOGIC;
24    SIGNAL hwf_start_r           : STD_LOGIC;
25    SIGNAL hwf_connectionsIn0_r  : uint8_t;
26    SIGNAL hwf_wanted1_r         : uint8_t;
27    SIGNAL hwf_learn0_r          : STD_LOGIC;
28    CONSTANT NeuralNetwork_id    : uint8_t := x"60";

29 BEGIN
30    NeuralNetwork_hwf : ENTITY work.NeuralNetwork(Behavioral)
31       PORT MAP(
32          clk            => skeleton_clock,
33          reset          => skeleton_reset,
34          connectionsIn  => hwf_connectionsIn0_r,
35          wanted         => hwf_wanted1_r,
36          connectionsOut => hwf_connectionsOut0_s,
37          learn          => hwf_learn0_r,
38          start          => hwf_start_r,
39          dataRdy        => hwf_dataRdy_s
40       );

41    NeuralNetwork_p : PROCESS (skeleton_clock, skeleton_reset)
42    BEGIN
43       IF skeleton_reset = '1' THEN
44          hwf_connectionsIn0_r <= (OTHERS => '0');
45          hwf_start_r          <= '0';
46          hwf_wanted1_r        <= (OTHERS => '0');
47          hwf_learn0_r         <= '0';
48       ELSIF rising_edge(skeleton_clock) THEN
49          IF skeleton_write = '1' THEN
50             skeleton_write_once_s <= '1';
51          END IF;
52          IF skeleton_write = '0' THEN
53             CASE to_integer(skeleton_address) IS
54                WHEN 0 =>
55                   hwf_connectionsIn0_r <= skeleton_data_in;
56                WHEN 1 =>
57                   hwf_wanted1_r <= skeleton_data_in;
58                WHEN 2 =>
59                   IF skeleton_write_once_s = '1' THEN
60                      hwf_start_r          <= skeleton_data_in(0);
61                      hwf_learn0_r         <= skeleton_data_in(1);
```

```
62                        skeleton_write_once_s <= '0';
63                     END IF;
64                  WHEN OTHERS =>
65                     -- Intentionally empty
66               END CASE;
67            ELSIF skeleton_read = '0' THEN
68               CASE to_integer(skeleton_address) IS
69                  WHEN 2 =>
70                     skeleton_data_out(0)          <= hwf_start_r;
71                     skeleton_data_out(1)          <= hwf_learn0_r;
72                     skeleton_data_out(7 DOWNTO 2) <= (OTHERS => '0');
73                  WHEN 3                                  =>
74                     skeleton_data_out <= hwf_connectionsOut0_s;
75                  WHEN 4 =>
76                     skeleton_data_out(0)          <= hwf_dataRdy_s;
77                     skeleton_data_out(7 DOWNTO 1) <= (OTHERS => '0');
78                  WHEN OTHERS                             =>
79                     skeleton_data_out <= NeuralNetwork_id;
80               END CASE;
81            END IF;
82         END IF;
83      END PROCESS;
84   END Behavioral;
```

The generic data interface required by the Elastic Node middleware consists of the connections `skeleton_data_in`, `skeleton_data_out`, `skeleton_address`, `skeleton_read` and `skeleton_write`. The addressing is done through the `case` switch, allocating different behaviour to every address. Additionally, a number of registers (e.g. `hwf_start_r` and `hwf_dataRdy_s`) are inserted transparently for convenience.

C.1. ANN Stub

The second part of the generated abstractions is the stub which allows a user to incorporate the hardware accelerator into their software projects. It consists of a header (shown in Listing C.2) and a source file (shown in Listing C.3). Again, the source file itself is not used directly, since the interface is exclusively provided by the API header.

The header provided in Listing C.2 creates the needed function definitions. It consists primarily of `set` and `query` functions for providing input to and retrieving output from the HWF.

Code Listing C.2.: Generated stub header for the ANN HWF

```
#ifndef NEURALNETWORK_H
#define NEURALNETWORK_H

void neuralnetwork_set_connectionsIn(uint8_t val);
void neuralnetwork_set_wanted(uint8_t val);
void neuralnetwork_set_start(uint8_t val);
void neuralnetwork_set_learn(uint8_t val);
uint8_t neuralnetwork_query_connectionsOut(void);
uint8_t neuralnetwork_query_dataRdy(void);
uint8_t neuralnetwork_query_id(void);
uint8_t neuralNetwork_hwf_is_deployed(void);

#endif /* NEURALNETWORK_H */
```

Lastly, the source of the stub can be seen in Listing C.3 where the memory-mapped interactions can be seen. Firstly, the relevant pointers are created to the external memory interface (the `USERLOGIC_OFFSET` constant is provided by the middleware and directs the interaction to the XMEM interface).

Code Listing C.3.: Generated stub source file for the ANN HWF

```
/*******************
 * External pointers:
 *******************/
static volatile uint8_t *const hwf_connectionsIn =
    (uint8_t *)(USERLOGIC_OFFSET + 0);

/* 1 byte [datatype size] offset of "connectionsIn" */
static volatile uint8_t *const hwf_wanted =
    (uint8_t *)(USERLOGIC_OFFSET + 1);

/* 1 byte [datatype size] offset of "wanted" */
static volatile uint8_t *const hwf_ctrl_in =
    (uint8_t *)(USERLOGIC_OFFSET + 2);

/* 1 byte [datatype size] offset of "hwf_ctrl_in" */
static volatile uint8_t *const hwf_connectionsOut =
    (uint8_t *)(USERLOGIC_OFFSET + 3);
```

```
15 /* 1 byte [datatype size] offset of "connectionsOut" */
16 static volatile uint8_t *const hwf_ctrl_out =
17     (uint8_t *)(USERLOGIC_OFFSET + 4);

18 /* 1 byte [datatype size] offset of "hwf_ctrl_out" */
19 static volatile uint8_t *const hwf_id =
20     (uint8_t *)(USERLOGIC_OFFSET + 5);

21 static const uint8_t STUB_ID =
22     96;

23 /*********************************
24  * HWF interface access functions:
25  *********************************/
26 void neuralnetwork_set_connectionsIn(uint8_t val)
27 {
28     *hwf_connectionsIn = val;
29 }

30 void neuralnetwork_set_wanted(uint8_t val)
31 {
32     *hwf_wanted = val;
33 }

34 void neuralnetwork_set_start(uint8_t val)
35 {
36     *hwf_ctrl_in |= ((val & 1) << 0);
37 }

38 void neuralnetwork_set_learn(uint8_t val)
39 {
40     *hwf_ctrl_in |= ((val & 1) << 1);
41 }

42 uint8_t neuralnetwork_query_connectionsOut(void)
43 {
44     return *hwf_connectionsOut;
45 }

46 uint8_t neuralnetwork_query_dataRdy(void)
```

```
{
    return (*hwf_ctrl_out & 1);
}

uint8_t neuralnetwork_query_id(void)
{
    return *hwf_id;
}

uint8_t neuralNetwork_hwf_is_deployed(void)
{
    return (STUB_ID == *hwf_id);
}
```

The remainder of the stub source is dedicated to transferring data and bit shifting based on the IDL. Some standard functions are also provided that return which HWF is currently deployed by checking the current ID.

Appendix D.

Evolution of Hardware Versions

A number of different versions of the Elastic Node hardware platform have been created, as will be shown here. Starting from humble beginnings using a breadboard (the infamous missing Elastic Node v1), our platform has seen much improvement over the various iterations. New features have been introduced, and more modern hardware has been incorporated – proving our platform to be easily adaptable to different hardware components.

D.1. Elastic Node v2

The first custom PCB version of the Elastic Node hardware platform (shown in Figure D.1) was based on the Spartan 6 LX9 and the Atmel ATMega64. It was a limited success as it offered limited heterogeneous functionality, but proved the potential of the design. It relied on indirect FPGA configuration, as there was no shared storage. This meant that each configuration had to be loaded using the MCU direct method shown in Table 7.3, leading to massive start-up times.

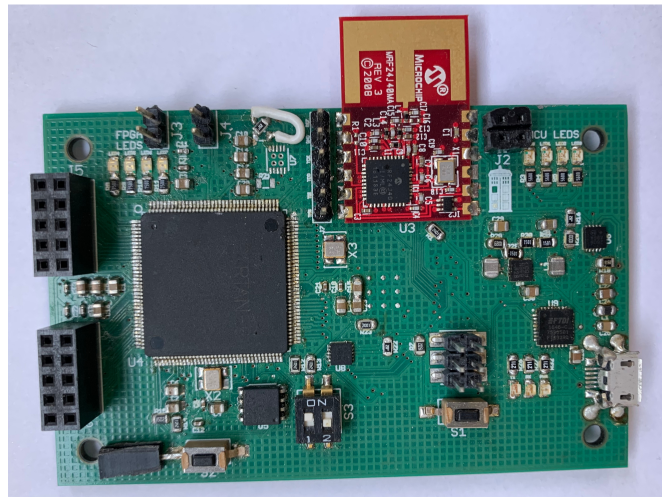


Figure D.1.: Elastic Node v2

D.2. Elastic Node v3

The next iteration visible in Figure D.2 was considerably more successful. Although based on the same main components, it introduced the current monitoring hardware discussed in Section 4.5.2. It also added a second flash chip (one connected to the MCU and one to the FPGA), which enabled the usage of an indirect flash programming method that involved a custom FPGA configuration that relayed incoming data to the flash via the PL of the FPGA.

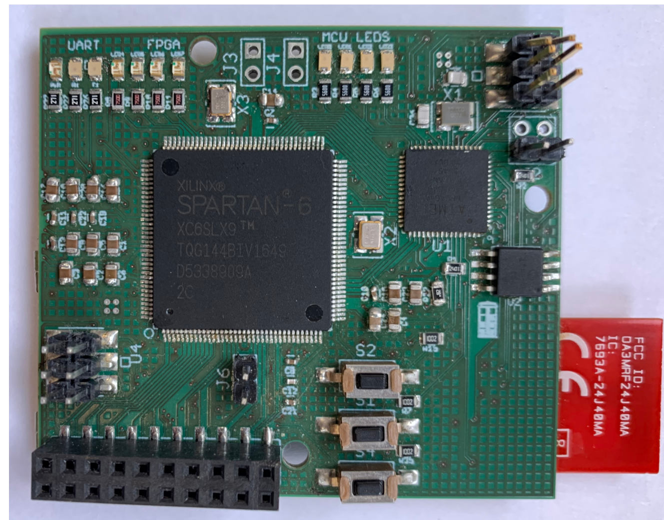


Figure D.2.: Elastic Node v3

This version of the platform was the first to be publicly demonstrated, proving its ability to outperform a server-based offloading approach to ANNs at PerCom 2018 [31].

D.3. Elastic Node v4

A much later version of the Elastic Node shown in Figure D.3 saw a major upgrade in the components used. It changed to the USB-capable AT90USB128 and the newer generation Spartan 7 LX15 and LX25. Apart from offering considerably more resources to applications, this showed the flexibility of the design to new components, as only a few middleware components (such as the FPGA-side reconfiguration logic) had to be updated. It also improved the electrical design and aesthetics of the board by upgrading from the Thin Quad Flat Pack (TQFP) package to the Ball Grid Array (BGA) for the FPGA. This also made the board much easier to assemble using either hot air or a soldering oven, simplifying its production in larger numbers.

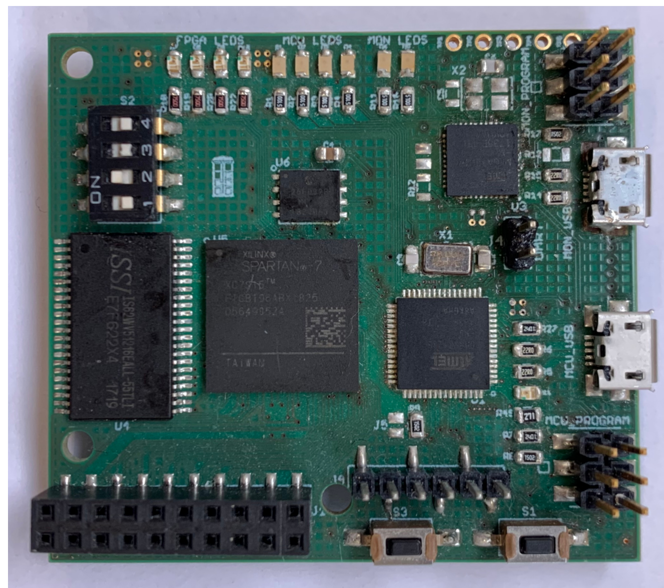


Figure D.3.: Elastic Node v4

This board was used to demo our platform at the ICAC 2019 conference [211], showing its ability to provide experimentation data while calculating CNNs.

D.4. ARM Elastic Node

With valuable assistance from both a student project group and Chao Qian, an ARM-based Elastic Node was developed in-house (Figure D.4). This device used a much more powerful MCU in the 32-bit Cortex M4 (LPC4088), and increased the package size of the FPGA (to the FGGA484) to support considerably larger FPGAs (e.g. the Spartan 7 LX50 and LX100).

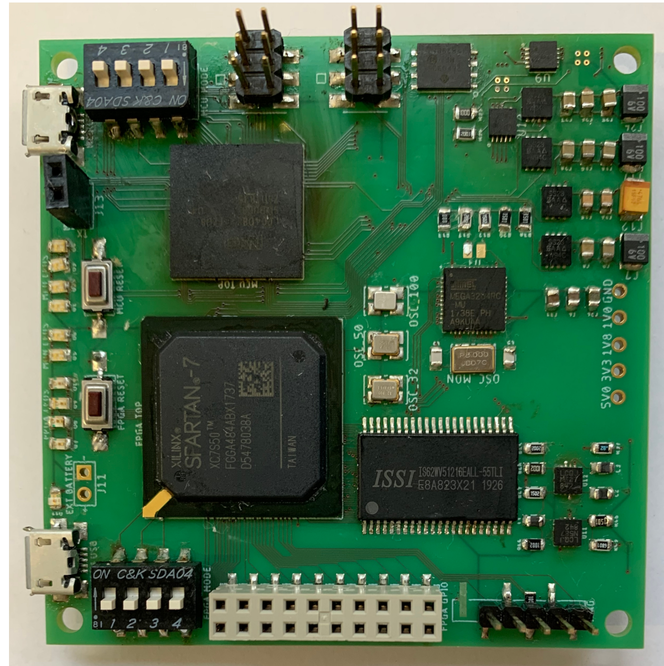


Figure D.4.: ARM-based Elastic Node

At the time of writing this version of the hardware platform is still under continuous software development to support the full functionality of the Elastic Node middleware, but its hardware is already fully functional.

D.5. ARM Elastic Node v2

This variant of the Elastic Node was developed and used primarily during the thesis of Chao Qian [191]. It features an even more powerful Cortex M4 MCU in the STM32F4, as well as a generally better electrical layout (e.g. using a ground plane for increased noise mitigation).

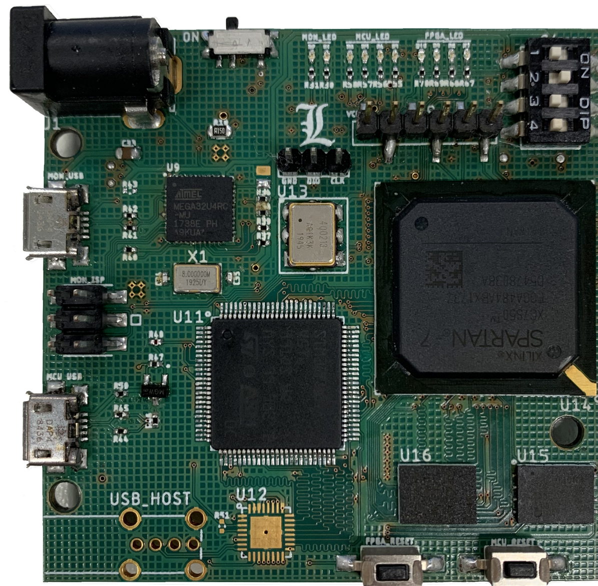


Figure D.5.: Variable frequency ARM Elastic Node

The main benefit of this iteration was that it featured a variable clock source controllable by the MCU. This provided the opportunity for experiments at varying clock rates without requiring hardware changes.

This long list of iterations of the Elastic Node proves the adaptability of the fundamental design. Throughout the different created versions, the hardware specifications have drastically changed – and new functionality has been added to it – but the core design remains the same. It provides insight into the full potential of our design, which will undoubtedly see further growth and development.

Bibliography

[1] Ken Addison. *ASUS ROG SWIFT PG27UQ*, 2018, https://pcper.com/2018/06/asus-rog-swift-pg27uq-27-4k-144hz-g-sync-monitor-true-hdr-arrives-on-the-desktop/2/ (last visited: 2020-11-12).

[2] Andreas Agne, Markus Happe, Ariane Keller, Enno Lubbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. *ReconOS: An operating system approach for reconfigurable computing.* IEEE Micro, 34(1):60–71, 2014. doi: 10.1109/MM.2013.110.

[3] Aamir Akbar and Peter R. Lewis. *Self-adaptive and self-aware mobile-cloud hybrid robotics.* International Conference on Internet of Things: Systems, Management and Security (IoTSMS), pages 262–267. IEEE, 2018. ISBN 9781538695852. doi: 10.1109/IoTSMS.2018.8554735.

[4] Juan Valverde Alcalá. *Run-Time Dynamically-Adaptable FPGA-Based Architecture for High-Performance Autonomous Distributed Systems.* PhD thesis, Universidad Politécnica de Madrid, 2015.

[5] Amazon Web Services. *FreeRTOS - Market leading RTOS*, 2021, https://www.freertos.org/ (last visited: 2021-11-13).

[6] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. *Fixed point optimization of deep convolutional neural networks for object recognition.* International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 1131–1135. IEEE, 2015. ISBN 9781467369978. doi: 10.1109/ICASSP.2015.7178146.

[7] Emekcan Aras, Stéphane Delbruel, Fan Yang, Wouter Joosen, and Danny Hughes. *Chimera: A Low-power Reconfigurable Platformfor Internet of Things.* ACM Transactions on Internet of Things, 2(2):1–25, may 2021. ISSN 2691-1914. doi: 10.1145/3440995.

[8] Arduino. *Arduino - Home*, 2021, https://www.arduino.cc/ (last visited: 2021-11-13).

[9] Arm. *big.LITTLE*, 2021, https://www.arm.com/why-arm/technologies/big-little (last visited: 2021-10-13).

[10] ARM Limited. *AMBA AXI and ACE Protocol Specification [White paper]*, 2011.

[11] Matej Artač, Matjaž Jogan, and Aleš Leonardis. *Incremental PCA for on-line visual learning and recognition.* International Conference on Pattern Recognition (ICPR), volume 16, pages 781–784, 2002. doi: 10.1109/icpr.2002.1048133.

[12] Ahmed Abdulwali Mohammed Haidar Al Asbahi, Feng Zhi Gang, Wasim Iqbal, Qaiser Abass, Muhammad Mohsin, and Robina Iram. *Novel approach of Principal Component Analysis method to assess the national energy performance via Energy Trilemma Index.* Energy Reports, 5:704–713, 2019. ISSN 23524847. doi: 10.1016/j.egyr.2019.06.009.

[13] Jesús Barba, Félix Jesús Villanueva, Manuel Abaldea, David Villa, Oscar Aceña, and Juan Carlos López. *Off-the-shelf embedded middleware solution for UAVs HW-SW platform development.* International Conference on Advanced Information Networking and Applications Workshops (WAINA), pages 815–820, 2016. ISBN 9781509018574. doi: 10.1109/WAINA.2016.70.

[14] Chloe M. Barnes, Anikó Ekárt, and Peter R. Lewis. *Social Action in Socially Situated Agents.* International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pages 97–106. IEEE, 2019. ISBN 9781728127316. doi: 10.1109/SASO.2019.00021.

[15] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. *BASE - a micro-broker-based middleware for pervasive computing.* International Conference on Pervasive Computing and Communications (PerCom), pages 443–451. IEEE, 2003. ISBN 0-7695-1893-1. doi: 10.1109/PERCOM.2003.1192769.

[16] Christian Becker, Jörg Hähner, and Sven Tomforde. *Flexibility in organic systems remarks on mechanisms for adapting system goals at runtime.* International Conference on Informatics in Control, Automation and Robotics (ICINCO), pages 287–292. Science and Technology Publications, 2012. ISBN 9789898565211. doi: 10.5220/0004121002870292.

[17] Shai Ben-David, Eyal Kushilevitz, and Yishay Mansour. *Online Learning versus Offline Learning.* Machine Learning, 29(1):45–63, 1997. ISSN 08856125. doi: 10.1023/A:1007465907571.

[18] Olivier Berder and Olivier Sentieys. *PowWow: Power Optimized Hardware/Software Framework for Wireless Motes.* International Conference on Architecture of Computing Systems (ARCS), pages 1–5. IEEE, 2010. ISBN 978-3-8007-3222-7.

[19] Neil Bergmann, John Alan Williams, and Peter Waldeck. *Egret: A flexible platform for real-time reconfigurable systems on chip.* The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), pages 300 – 303. CSREA Press, 2003.

[20] A. Bernardini and S. De Fina. *A neural network to approximate nonlinear functions.* Midwest Symposium on Circuits and Systems, pages 545–548. IEEE, 1991. ISBN 0780306201. doi: 10.1109/MWSCAS.1991.252103.

[21] Andreas Bernauer, Johannes Zeppenfeld, Oliver Bringmann, Andreas Herkersdorf, and Wolfgang Rosenstiel. *Combining software and hardware LCS for lightweight on-chip learning.* Distributed, Parallel and Biologically Inspired Systems (DIPES), volume 329, pages 278–289. Springer, 2010. ISBN 9783642152337. doi: 10.1007/978-3-642-15234-4_27.

[22] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. *Fog computing and its role in the internet of things.* MCC workshop on Mobile cloud computing (MCC), pages 13–16. Association for Computing Machinery, 2012. ISBN 9781450315197. doi: 10.1145/2342509.2342513.

[23] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics.* Studies in Computational Intelligence, 546:169–186, 2014. ISSN 1860949X. doi: 10.1007/978-3-319-05029-4.

[24] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. *MAVBench: Micro aerial vehicle benchmarking.* Annual International Symposium on Microarchitecture (MICRO). IEEE, 2018. ISBN 9781538662403. doi: 10.1109/MICRO.2018.00077.

[25] Abdelmajid Bouajila, Johannes Zeppenfeld, W Stechele, Andreas Bernauer, O Bringmann, W Rosenstiel, and A Herkersdorf. Autonomic System on Chip Platform. Organic Computing, pages 413–425. Springer, 2011. doi: 10.1007/978-3-0348-0130-0_27.

[26] Jayson Boubin, John Chumley, Christopher Stewart, and Sami Khanal. *Autonomic Computing Challenges in Fully Autonomous Precision Agriculture.* International Conference on Autonomic Computing (ICAC), number June, pages 11–17. IEEE, 2019. ISBN 9781728124117. doi: 10.1109/ICAC.2019.00012.

[27] Jayson G. Boubin, Naveen T.R. Babu, Christopher Stewart, John Chumley, and Shiqi Zhang. *Managing edge resources for fully autonomous aerial systems.* Symposium on Edge Computing (SEC), pages 74–87. Association for Computing Machinery, 2019. ISBN 9781450367332. doi: 10.1145/3318216.3363306.

[28] Jürgen Branke, Moez Mnif, Christian Müller-Schloer, Holger Prothmann, Urban Richter, Fabian Rochner, and Hartmut Schmeck. *Organic Computing - Addressing complexity by controlled self-organization.* International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), pages 185–191. IEEE, 2006. ISBN 0769530710. doi: 10.1109/ISoLA.2006.19.

[29] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. *Language models are few-shot learners.* Advances in Neural Information Processing Systems, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[30] Robert Brzoza-Woch and Piotr Nawrocki. *Reconfigurable FPGA-based embedded Web services as distributed computational nodes.* Federated Conference on Computer Science and Information Systems, volume 6, pages 159–164. PTI, 2015. doi: 10.15439/2015F37.

[31] Alwyn Burger and Gregor Schiele. *Demo Abstract: Deep Learning on an Elastic Node for the Internet of Things.* International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pages 555–557. IEEE, 2018. ISBN 9781538632277. doi: 10.1109/PERCOMW.2018.8480160.

[32] Alwyn Burger, Christopher Cichiwskyj, and Gregor Schiele. *Elastic Nodes for the Internet of Things: A Middleware-Based Approach.* International Conference on Autonomic Computing (ICAC), pages 73–74. IEEE, 2017. ISBN 978-1-5386-1762-5. doi: 10.1109/ICAC.2017.27.

[33] Alwyn Burger, Christopher Cichiwskyj, Stephan Schmeißer, and Gregor Schiele. *The Elastic Internet of Things - A Platform for Self-Integrating and Self-Adaptive IoT-Systems with Support for Embedded Adaptive Hardware.* Future Generation Computer Systems, 113:607–619, 2020. doi: 10.1016/j.future.2020.07.035.

[34] Alwyn Burger, David King, and Gregor Schiele. *Reconfigurable embedded devices using reinforcement learning to develop action-policies.* International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, 2020. ISBN 9781728172774. doi: 10.1109/ACSOS49614.2020.00046.

[35] Alwyn Burger, Chao Qian, Gregor Schiele, and Domenik Helms. *An Embedded CNN Implementation for On-Device ECG Analysis.* International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). IEEE, 2020. doi: 10.1109/PerComWorkshops48775.2020.9156260.

[36] Alwyn Burger, Patrick Urban, Jayson Boubin, and Gregor Schiele. *An Architecture for Solving the Eigenvalue Problem on Embedded FPGAs.* International Conference on Architecture of Computing Systems (ARCS), pages 32–43. Springer, 2020. doi: 10.1007/978-3-030-52794-5_3.

[37] Alwyn Burger, Gregor Schiele, and David W King. *Developing Action Policies with Q-Learning and Shallow Neural Networks on Reconfigurable Embedded Devices.* ACM Transactions on Autonomous and Adaptive Systems, 15(4):1–25, 2021. doi: 10.1145/3487920.

[38] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. *Economic models for resource management and scheduling in grid computing.* Concurrency Computation Practice and Experience, 14(13-15):1507–1542, 2002. ISSN 15320626. doi: 10.1002/cpe.690.

[39] Olivier Chapelle, Mingmin Chi, and Alexander Zien. *A continuation method for semi-supervised SVMs.* International Conference on Machine Learning (ICML), volume 148, pages 185–192. Association for Computing Machinery, 2006. ISBN 1595933832. doi: 10.1145/1143844.1143868.

[40] Meng Hsi Chen, Min Dong, and Ben Liang. *Multi-user Mobile Cloud Offloading Game with Computing Access Point.* International Conference on Cloud Networking, CloudNet 2016, pages 64–69. IEEE, 2016. ISBN 9781509050932. doi: 10.1109/CloudNet.2016.52.

[41] Weiwei Chen, Dong Wang, and Keqin Li. *Multi-user Multi-task Computation Offloading in Green Mobile Edge Cloud Computing.* IEEE Transactions on Services Computing, 12(5):1–13, 2019. ISSN 19391374. doi: 10.1109/TSC.2018.2826544.

[42] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Mehdi Bennis. *Performance Optimization in Mobile-Edge Computing via Deep Reinforcement Learning.* Vehicular Technology Conference (VTC-Fall), pages 1–6. IEEE, 2018. ISBN 9781538663585. doi: 10.1109/VTCFall.2018.8690980, `http://arxiv.org/abs/1804.00514`.

[43] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Mehdi Bennis. *Optimized Computation Offloading Performance in Virtual Edge Computing Systems via Deep Reinforcement Learning.* IEEE Internet of Things Journal, 6(3):1–31, 2019. arXiv ID: 1805.06146. ISSN 23274662. doi: 10.1109/JIOT.2018.2876279.

[44] Xu Chen. *Decentralized computation offloading game for mobile cloud computing.* IEEE Transactions on Parallel and Distributed Systems, 26(4):974–983, apr 2015. ISSN 10459219. doi: 10.1109/TPDS.2014.2316834.

[45] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. *Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing.* IEEE/ACM Transactions on Networking, 24(5):2795–2808, 2016. arXiv ID: 1510.00888. ISSN 10636692. doi: 10.1109/TNET.2015.2487344.

[46] Long Cheng, Jianwei Niu, Chengwen Luo, Lei Shu, Linghe Kong, Zhiwei Zhao, and Yu Gu. *Towards minimum-delay and energy-efficient flooding in low-duty-cycle wireless sensor networks.* Computer Networks: The International Journal of Computer and Telecommunications Networking, 134(C):66–77, 2018. ISSN 13891286. doi: 10.1016/j.comnet.2018.01.012.

[47] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning phrase representations using RNN encoder-decoder for statistical machine translation.* Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1724–1734. Association for Computing Machinery, 2014. ISBN 9781937284961. doi: 10.3115/v1/d14-1179.

[48] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. *CloneCloud: Elastic Execution Between Mobile Device and Cloud.* EuroSys, pages 301–314. Association for Computing Machinery, 2011. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966473.

[49] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. *Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?* Proceedings of the Annual International Symposium on Microarchitecture, MICRO, pages 225–236, 2010. ISBN 9780769542997. doi: 10.1109/MICRO.2010.36.

[50] Christopher Cichiwskyj and Gregor Schiele. *Using field-programmable gate arrays for learning non player characters.* International Workshop on Massively Multiuser Virtual Environments (MMVE), pages 1–2. Association for Computing Machinery, 2016. ISBN 9781450343589. doi: 10.1145/2910659.2910662.

[51] Christopher Cichiwskyj and Gregor Schiele. *Temporal Accelerators: Unleashing the Potential of Embedded FPGAs.* JUCS - Journal of Universal Computer Science, 27(11):1174–1192, 2021. ISSN 0948-695X. doi: 10.3897/jucs.77247.

[52] Christopher Cichiwskyj, Chao Qian, and Gregor Schiele. Time to Learn: Temporal Accelerators as an Embedded Deep Neural Network Platform. IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning, pages 256–267. Springer, 2020. ISBN 9783030667696. doi: 10.1007/978-3-030-66770-2_19.

[53] Christopher Cichiwskyj, Stephan Schmeißer, Chao Qian, Lukas Einhaus, Christopher Ringhofer, and Gregor Schiele. *Elastic AI: system support for adaptive machine learning in pervasive computing systems.* CCF Transactions on Pervasive Computing and Interaction, 3(3):300–328, 2021. ISSN 25245228. doi: 10.1007/s42486-021-00070-6, `https://doi.org/10.1007/s42486-021-00070-6`.

[54] Colin Clausen, Simon Reichhuber, Ingo Thomsen, and Sven Tomforde. *Improvements to increase the efficiency of the AlphaZero algorithm: A case study in the Game 'Connect 4'.* International Conference on Agents and Artificial Intelligence (ICAART), volume 2, pages 803–811. SciTePress, 2021. ISBN 9789897584848. doi: 10.5220/0010245908030811.

[55] Jason Cong and Yi Zou. *FPGA-based hardware acceleration of lithographic aerial image simulation.* ACM Transactions on Reconfigurable Technology and Systems, 2(3):1–29, 2009. ISSN 19367406. doi: 10.1145/1575774.1575776.

[56] Francesco Conti, Daniele Pajossit, Andrea Marongiu, Davide Rossi, and Luca Benini. *Enabling the Heterogeneous Accelerator Model on Ultra-Low Power Microcontroller Platforms.* Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1201–1206. IEEE, 2016. ISBN 9783981537079.

[57] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Gian Pietro Picco, and Stefanos Zachariadis. *The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems.* International Symposium on Personal, Indoor and Mobile Radio Communications, pages 806–810. IEEE, 2005. ISBN 978-3-8007-29. doi: 10.1109/PIMRC.2005.1651554.

[58] Eduardo Cuervo, Aruna Balasubramanian, and Dae-ki Cho. *MAUI: making smartphones last longer with code offload.* International Conference on Mobile Systems, Applications, and Services (MobiSys), pages 49–62. Association for Computing Machinery, 2010. ISBN 9781605589855. doi: 10.1145/1814433.1814441.

[59] Zhihua Cui and Xiaozhi Gao. *Theory and applications of swarm intelligence.* Neural Computing and Applications, 21(2):205–206, 2012. ISSN 09410643. doi: 10.1007/s00521-011-0523-8.

[60] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. *From OpenCL to high-performance hardware on FPGAS.* International Conference on Field Programmable Logic and Applications (FPL), pages 531–534. IEEE, aug 2012. ISBN 978-1-4673-2256-0. doi: 10.1109/FPL.2012.6339272.

[61] Martin Danek, Jiri Kadlec, Roman Bartosinski, and Lukas Kohout. *Increasing the level of abstraction in FPGA-based designs.* International Conference on Field Programmable Logic and Applications (FPL), pages 5–10. IEEE, 2008. ISBN 9781424419616. doi: 10.1109/FPL.2008.4629899.

[62] Mirko D'Angelo, Simos Gerasimou, Sona Ghahremani, Johannes Grohmann, Ingrid Nunes, Evangelos Pournaras, and Sven Tomforde. *On learning in collective self-adaptive systems: State of practice and a 3D framework.* International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pages 13–24. IEEE, 2019. ISBN 9781728133683. doi: 10.1109/SEAMS.2019.00012.

[63] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. *DIANNE: Distributed Artificial Neural Networks for the Internet of Things*. Workshop on Middleware for Context-Aware Applications in the IoT (M4IoT), pages 19–24. Association for Computing Machinery, 2015. ISBN 9781450337311. doi: 10.1145/2836127.2836130.

[64] Bernhard Dieber, Jennifer Simonjan, Lukas Esterle, Bernhard Rinner, Georg Nebehay, Roman Pflugfelder, and Gustavo Javier Fernandez. *Ella: Middleware for multi-camera surveillance in heterogeneous visual sensor networks*. International Conference on Distributed Smart Cameras (ICDSC), pages 1–6. IEEE, 2013. doi: 10.1109/ICDSC.2013.6778223.

[65] Thinh Quang Dinh, Jianhua Tang, Quang Duy La, and Tony Q.S. Quek. *Offloading in Mobile Edge Computing: Task Allocation and Computational Frequency Scaling*. Transactions on Communications, 65(8):3571–3584, 2017. ISSN 15580857. doi: 10.1109/TCOMM.2017.2699660.

[66] R. Döhler. *Squared givens rotation*. IMA Journal of Numerical Analysis, 11(1): 1–5, 1991. ISSN 02724979. doi: 10.1093/imanum/11.1.1.

[67] Jinwen Du. *Noise Mitigation for CNN Classifiers in Embedded Environments*. Master's thesis, University of Duisburg-Essen, 2020.

[68] Nikil Dutt, Carlo S Regazzoni, Bernhard Rinner, and Xin Yao. *Self-Awareness for Autonomous Systems*. Proceedings of the IEEE, 108(7):971–975, jul 2020. ISSN 0018-9219. doi: 10.1109/JPROC.2020.2990784, https://ieeexplore.ieee.org/document/9120415/.

[69] R Eberhart and J Kennedy Sixth. *A new optimizer using particle swarm theory*. International Symposium on Micro Machine and Human Science (MHS), pages 39–43. IEEE, 1995. ISBN 0780326768. doi: 10.1109/MHS.1995.494215.

[70] Lukas Einhaus, Chao Qian, Christopher Ringhofer, and Gregor Schiele. *Towards Precomputed 1D-Convolutional Layers for Embedded FPGAs*. Machine Learning and Principles and Practice of Knowledge Discovery in Databases, pages 327–338, Cham, 2021. Springer International Publishing. ISBN 978-3-030-93736-2. doi: 10.1007/978-3-030-93736-2_25.

[71] Sven Eisenhardt, Thomas Schweizer, Julio Oliveira Filho, Tommy Kuhn, and Wolfgang Rosenstiel. Evaluation and Design Methods for Processor-Like Reconfigurable Architectures. Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications, pages 95–116. Springer Netherlands, Dordrecht, 2010. ISBN 978-90-481-3485-4. doi: 10.1007/978-90-481-3485-4_5.

[72] Andreas Engel and Andreas Koch. *Heterogeneous Wireless Sensor Nodes that Target the Internet of Things*. IEEE Micro, 36(6):8–15, 2016. ISSN 02721732. doi: 10.1109/MM.2016.100.

[73] Andreas Engel, Andreas Koch, and Thomas Siebel. *A heterogeneous system architecture for low-power wireless sensor nodes in compute-intensive distributed applications*. Local Computer Networks Conference Workshops (LCN Workshops), pages 636–644. IEEE, 2015. ISBN 9781467367738. doi: 10.1109/LCNW.2015.7365908.

[74] Ambrose Finnerty and Hervé Ratigner. *Reduce Power and Cost by Converting from Floating Point to Fixed Point Introduction [White Paper]*, 2017.

[75] FiPS. *FiPS - Developing Hardware and Design Methodologies for Heterogeneous Low Power Field Programmable Servers*, 2015, `https://www.fips-project.eu/` (last visited: 2015-12-28).

[76] Huber Flores, Pan Hui, Sasu Tarkoma, Yong Li, Satish Srirama, and Rajkumar Buyya. *Mobile code offloading: From concept to practice and beyond*. IEEE Communications Magazine, 53(3):80–88, 2015. ISSN 01636804. doi: 10.1109/MCOM.2015.7060486.

[77] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. *Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis*. International Conference on Embedded and Ubiquitous Computing, pages 120–129. IEEE, 2014. doi: 10.1109/EUC.2014.26.

[78] Nicolas Frick. *FPGA Energy Monitoring and Control through Power State Optimization*. Bachelor's thesis, University of Duisburg-Essen, 2019.

[79] Walter Gander. *Algorithms for the QR decomposition*. Research Report, 80(2):1251–1268, 1980.

[80] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. *Edge-centric Computing*. SIGCOMM Computer Communication Review, volume 45, pages 37–42. Association for Computing Machinery, 2015. doi: 10.1145/2831347.2831354.

[81] Hasan Genc, Yazhou Zu, Ting Wu Chin, Matthew Halpern, and Vijay Janapa Reddi. *Flying IoT: Toward Low-Power Vision in the Sky*. IEEE Micro, 37(6):40–51, 2017. ISSN 02721732. doi: 10.1109/MM.2017.4241339.

[82] Yoav Goldberg. *Neural Network Methods for Natural Language Processing*. Synthesis Lectures on Human Language Technologies, 10(1):1–309, 2017. ISSN 19474040. doi: 10.2200/S00762ED1V01Y201703HLT037.

[83] Juan A. Gomez-Pulido, Miguel A. Vega-Rodriguez, Juan M. Sanchez-Perez, Silvio Priem-Mendes, and Vitor Carreira. *Accelerating floating-point fitness functions in evolutionary algorithms: A FPGA-CPU-GPU performance comparison*. Genetic Programming and Evolvable Machines, 12(4):403–427, 2011. ISSN 13892576. doi: 10.1007/s10710-011-9137-2.

[84] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 1. MIT press, 2016. ISBN 9780262035613.

[85] Google. *TensorFlow*, 2021, `https://www.tensorflow.org/` (last visited: 2021-11-09).

[86] Root Gorelick, Susan M. Bertram, Peter R. Killeen, and Jennifer H. Fewell. *Normalized mutual entropy in biology: Quantifying division of labor*. American Naturalist, 164(5):677–682, 2004. ISSN 00030147. doi: 10.1086/424968.

[87] René Griessl, Meysam Peykanu, Jens Hagemeyer, Mario Porrmann, Stefan Krupop, Micha vor dem Berge, Thomas Kiesel, and Wolfgang Christmann. *A Scalable Server Architecture for Next-Generation Heterogeneous Compute Clusters*. International Conference on Embedded and Ubiquitous Computing, pages 146–153. IEEE, 2014. doi: 10.1109/EUC.2014.29.

[88] Yu Gu and Tian He. *Dynamic switching-based data forwarding for low-duty-cycle wireless sensor networks*. IEEE Transactions on Mobile Computing, 10(12): 1741–1754, 2011. ISSN 15361233. doi: 10.1109/TMC.2010.266.

[89] Jorge E. Guerrero-Ramírez, Jaime Velasco-Medina, and Julio C. Arce. *Hardware design of an eigensolver based on the QR method*. Analog Integrated Circuits and Signal Processing, 82(1):125–134, 2014. ISSN 15731979. doi: 10.1007/s10470-014-0445-3.

[90] Zakarya Guettatfi, Philipp Hubner, Marco Platzner, and Bernhard Rinner. *Computational self-awareness as design approach for visual sensor nodes*. International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), pages 1–8. IEEE, 2017. ISBN 9781538633441. doi: 10.1109/ReCoSoC.2017.8016147.

[91] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. *Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing*. International Conference on Computer Communications (INFOCOM), pages 1–9. IEEE, 2016. ISBN 9781467399531. doi: 10.1109/INFOCOM.2016.7524497.

[92] Lars Kai Hansen and Peter Salamon. *Neural Network Ensembles*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(10):993–1001, 1990. ISSN 01628828. doi: 10.1109/34.58871.

[93] Markus Happe, Enno Lübbers, and Marco Platzner. *A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking*. Journal of Real-Time Image Processing, 8(1):95–110, 2013. ISSN 18618200. doi: 10.1007/s11554-011-0212-y.

[94] Mark Harris. *Mixed-Precision Programming with CUDA 8*, 2016, `https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/` (last visited: 2021-11-06).

[95] Hado Van Hasselt. *Double Q-learning*. International Conference on Neural Information Processing Systems (NIPS), pages 2613–2621. Association for Computing Machinery, 2010. ISBN 9781617823800.

[96] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, Andreas Herkersdorf, Rüdiger Kapitza, Daniel Lohmann, Peter Marwedel, Marco Platzner, Wolfgang Rosenstiel, Ulf Schlichtmann, Olaf Spinczyk, Mehdi Tahoori, Jürgen Teich, Norbert Wehn, and Hans Joachim Wunderlich. *Design and architectures for dependable embedded systems*. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), number October, pages 69–78. IEEE, 2011. ISBN 9781450307154. doi: 10.1145/2039370.2039384.

[97] M Henning. *A New Approach to Object-Oriented Middleware*. IEEE Internet Computing, 8(1):66–75, 2004. doi: 10.1109/MIC.2004.1260706.

[98] Sascha Christian Hevelke. *An Approach for Efficient Runtime Self-Configuration for Embedded Reconfigurable Platforms*. Master's thesis, University of Duisburg-Essen, 2016.

[99] Stephen Hilt, Vladimir Kropotov, Fernando Mercês, Mayra Rosario, and David Sancho. *The Internet of Things in the Cybercrime Underground*, 2019, `https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/the-internet-of-things-in-the-cybercrime-underground` (last visited: 2021-12-21).

[100] John H Holland. Adaptation. Progress in Theoretical Biology, pages 263–293. Academic Press, 1976. ISBN 978-0-12-543104-0. doi: 10.1016/B978-0-12-543104-0.50012-3.

[101] Markus Horstmann and Mary Kirtland. *DCOM architecture - Software Toolbox*, 1997, `https://www.softwaretoolbox.com/dcom/DCOMArchitecture.pdf` (last visited: 2021-12-21).

[102] Michael Hosemann and Gerhard P. Fettweis. *On enhancing SIMD-controlled DSPs for performing recursive filtering.* Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 43:125–142, 2006. ISSN 13875485. doi: 10.1007/s11265-006-7266-2.

[103] Lingkon Hossain. *Analysis and Design of Audio Capturing Solutions for Low-Power Embedded Systems.* Master's thesis, University of Duisburg-Essen, 2019.

[104] Liang Huang, Xu Feng, Anqi Feng, Yupin Huang, and Li Ping Qian. *Distributed Deep Learning-based Offloading for Mobile Edge Computing Networks.* Mobile Networks and Applications, 2018. ISSN 15728153. doi: 10.1007/s11036-018-1177-x.

[105] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. *Deep Reinforcement Learning for Online Offloading in Wireless Powered Mobile-Edge Computing Networks.* IEEE Transactions on Mobile Computing, 19(11):1–24, 2020. doi: 10.1109/TMC.2019.2928811.

[106] Muhammad Imran, Khurram Shahzad, Naeem Ahmad, Mattias O'Nils, Najeem Lawal, and Bengt Oelmann. *Energy-Efficient SRAM FPGA-Based Wireless Vision Sensor Node: SENTIOF-CAM.* IEEE Transactions on Circuits and Systems for Video Technology, 24(12):2132–2143, dec 2014. ISSN 1051-8215. doi: 10.1109/TCSVT.2014.2330660.

[107] Intel. *Avalon Interface Specifications [White paper]*, 2013, `https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html` (last visited: 2021-12-21).

[108] Magnus Irestig, Niklas Hallberg, Henrik Eriksson, and Toomas Timpka. *Peer-to-peer computing in health-promoting voluntary organizations: A system design analysis.* Journal of Medical Systems, 29(5):425–440, 2005. ISSN 01485598. doi: 10.1007/s10916-005-6100-x.

[109] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. *RIFFA 2.1: A reusable integration framework for FPGA accelerators.* ACM Transactions on Reconfigurable Technology and Systems, 8(4):1–23, 2015. ISSN 19367414. doi: 10.1145/2815631.

[110] Vidit Jain and Erik Learned-Miller. FDDB: A Benchmark for Face Detection in Unconstrained Settings. Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[111] Richard Jiang and Danny Crookes. *Shallow Unorganized Neural Networks Using Smart Neuron Model for Visual Perception.* IEEE Access, 7:152701–152714, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2946422.

[112] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. *Reinforcement Learning: A Survey.* Journal of Artificial Intelligence Research, 4(1):237–285, 1996. doi: 10.5555/1622737.1622748.

[113] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. *A review on genetic algorithm: past, present, and future*, volume 80. Multimedia Tools and Applications, 2021. ISBN 1104202010139. doi: 10.1007/s11042-020-10139-6.

[114] Richard Kavanagh, Karim Djemame, Jorge Ejarque, Rosa M. Badia, and David Garcia-Perez. *Energy-Aware Self-Adaptation for Application Execution on Heterogeneous Parallel Architectures.* IEEE Transactions on Sustainable Computing, 5 (1):81–94, 2020. ISSN 23773782. doi: 10.1109/TSUSC.2019.2912000.

[115] James Kennedy. Swarm Intelligence. Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies, pages 187–219. Springer, Boston, MA, 2006. ISBN 978-0-387-27705-9. doi: 10.1007/0-387-27705-6_6.

[116] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing.* Computer, 36(1):41–50, 2003. doi: 10.1109/MC.2003.1160055.

[117] Anam Khalid. *Design and Implementation of an optimized SDRAM interface in Verilog.* Master's thesis, University of Duisburg-Essen, 2018.

[118] David King and Gilbert Peterson. *The Emergence of Division of Labor in Multi-Agent Systems.* International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pages 107–116. IEEE, 2019. ISBN 9781728127316. doi: 10.1109/SASO.2019.00022.

[119] David King, Lukas Esterle, and Gilbert L. Peterson. *Entropy-based team self-organization with signal suppression.* The Conference on Artificial Life (ALIFE), pages 145–152. Massachusetts Institute of Technology, 2019. doi: 10.1162/isal_a_00154.

[120] Robert Kirchgessner, Alan D George, and Greg Stitt. *Low-Overhead FPGA Middleware for Application Portability and Productivity.* ACM Transactions on Reconfigurable Technology and Systems, 8(4):1–22, 2015. ISSN 1936-7406. doi: 10.1145/2746404.

[121] Uday A. Korat and Amirhossein Alimohammad. *A Reconfigurable Hardware Architecture for Principal Component Analysis.* Circuits, Systems, and Signal Processing, 38(5):2097–2113, 2019. ISSN 15315878. doi: 10.1007/s00034-018-0953-y.

[122] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. *ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading.* International Conference on Computer Communications

(INFOCOM), pages 945–953. IEEE, 2012. ISBN 9781467307758. doi: 10.1109/INFCOM.2012.6195845.

[123] Yana Esteves Krasteva, Jorge Portilla, Jose María Carnicer, Eduardo de la Torre, and Teresa Riesgo. *Remote HW-SW reconfigurable wireless sensor nodes*. Annual Conference of the IEEE Industrial Electronics Society (IECON), pages 2483–2488. IEEE, 2008. ISBN 9781424417667. doi: 10.1109/IECON.2008.4758346.

[124] Christian Krupitzer, Felix Maximilian Roth, Martin Pfannemuller, and Christian Becker. *Comparison of approaches for self-improvement in self-adaptive systems*. International Conference on Autonomic Computing (ICAC), pages 308–314. IEEE, 2016. ISBN 9781509016532. doi: 10.1109/ICAC.2016.18.

[125] Karthik Kumar and Yung Hsiang Lu. *Cloud computing for mobile users: Can offloading computation save energy?* Computer, 43(4):51–56, 2010. ISSN 00189162. doi: 10.1109/MC.2010.98.

[126] Karthik Kumar, Jibang Liu, Yung Hsiang Lu, and Bharat Bhargava. *A survey of computation offloading for mobile systems*. Mobile Networks and Applications, 18 (1):129–140, 2013. ISSN 1383469X. doi: 10.1007/s11036-012-0368-0.

[127] Philippe Lalanda, Julie A McCann, and Ada Diaconescu. *Autonomic computing: principles, design and implementation*. Springer Science & Business Media, 2013. ISBN 978-1-4471-5007-7.

[128] Nicholas D. Lane and Petko Georgiev. *Can Deep Learning Revolutionize Mobile Sensing?* International Workshop on Mobile Computing Systems and Applications (HotMobile), pages 117–122. Association for Computing Machinery, 2015. ISBN 9781450333917. doi: 10.1145/2699343.2699349.

[129] Xuan Sang Le, Jean-Christophe Le Lann, Loic Lagadec, Luc Fabresse, Noury Bouraqadi, and Jannik Laval. *CaRDIN: An Agile Environment for Edge Computing on Reconfigurable Sensor Networks*. International Conference on Computational Science and Computational Intelligence (CSCI), number 2, pages 168–173. IEEE, 2016. ISBN 978-1-5090-5510-4. doi: 10.1109/CSCI.2016.0039.

[130] Sangil Lee, Changwook Son, and Hyojae Jang. *Distributed and parallel real-Time control system equipped FPGA-Zynq and EPICS middleware*. Conference on Real Time (RT), pages 1–4. IEEE, 2016. ISBN 9781509020140. doi: 10.1109/RTC.2016.7543117.

[131] Gregory Lento. *Optimizing Performance with Intel Advanced Vector Extensions [White Paper]*, 2014, `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf` (last visited: 2021-12-22).

[132] Sam Leroux, Steven Bohez, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. *Multi-fidelity matryoshka neural networks for constrained IoT devices.* International Joint Conference on Neural Networks (IJCNN), pages 1305–1309. IEEE, 2016. ISBN 9781509006199. doi: 10.1109/IJCNN.2016.7727348.

[133] Peter R. Lewis, Arjun Chandra, Shaun Parsons, Edward Robinson, Kyrre Glette, Rami Bahsoon, Jim Torresen, and Xin Yao. *A survey of self-awareness and its application in computing systems.* Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), pages 102–107. IEEE, 2011. ISBN 9780769545455. doi: 10.1109/SASOW.2011.25.

[134] Peter R. Lewis, Lukas Esterle, Arjun Chandra, Bernhard Rinner, Jim Torresen, and Xin Yao. *Static, Dynamic, and Adaptive Heterogeneity in Distributed Smart Camera Networks.* ACM Transactions on Autonomous and Adaptive Systems, 10 (2):1–30, 2015. doi: 10.1145/2764460.

[135] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. *Deep reinforcement learning based computation offloading and resource allocation for MEC.* IEEE Wireless Communications and Networking Conference, WCNC, 2018-April:1–6, 2018. ISSN 15253511. doi: 10.1109/WCNC.2018.8377343.

[136] Ning Li, Jose Fernan Martinez-Ortega, and Gregorio Rubio. *Distributed joint offloading decision and resource allocation for multi-user mobile edge computing: A game theory approach.* arXiv preprint, may 2018. arXiv ID: 1805.02182. ISSN 23318422.

[137] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. *Fixed point quantization of deep convolutional networks.* 33rd International Conference on Machine Learning, ICML 2016, volume 6, 2016.

[138] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B. Letaief. *Delay-optimal computation task scheduling for mobile-edge computing systems.* International Symposium on Information Theory (ISIT), pages 1451–1455. IEEE, 2016. ISBN 9781509018062. doi: 10.1109/ISIT.2016.7541539.

[139] Yu Liu, Qimei Cui, Jian Zhang, Yu Chen, and Yanzhao Hou. *An Actor-Critic Deep Reinforcement Learning Based Computation Offloading for Three-Tier Mobile Computing Networks.* International Conference on Wireless Communications and Signal Processing (WCSP). IEEE, 2019. ISBN 9781728135557. doi: 10.1109/WCSP.2019.8927911.

[140] Chris Lomont. *Fast inverse square root*, 2002, `https://www.lomont.org/papers/2003/InvSqrt.pdf` (last visited: 2021-12-22).

[141] Tom H. Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. *Fog Computing: Focusing on Mobile Users at the Edge.* arXiv preprint, pages 1–11, 2015. arXiv ID: 1502.01815.

[142] Enno Lübbers and Marco Platzner. *Reconos: An RTOS supporting hard- and software threads.* International Conference on Field Programmable Logic and Applications (FPL), pages 441–446. IEEE, 2007. doi: 10.1109/FPL.2007.4380686.

[143] Enno Lübbers and Marco Platzner. *ReconOS: Multithreaded Programming for Reconfigurable Computers.* ACM Transactions on Embedded Computing Systems, 9(1):1–33, 2009. doi: 10.1145/1596532.1596540.

[144] Xiao Ma, Chuang Lin, Xudong Xiang, and Congjie Chen. *Game-theoretic Analysis of Computation Offloading for Cloudlet-based Mobile Cloud Computing.* International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), pages 271–278. Association for Computing Machinery, 2015. ISBN 9781450337625. doi: 10.1145/2811587.2811598.

[145] Hr Flores Macario, Satish Srirama, Huber Flores, and Satish Srirama. *Adaptive code offloading for mobile cloud applications: exploiting fuzzy sets and evidence-based learning.* Workshop on Mobile Cloud Computing and Services (MCS), pages 9–16. Association for Computing Machinery, 2013. doi: 10.1145/2497306.2482984.

[146] Frederic Magoules, Jie Pan, Kiat-An Tan, and Abhinit Kumar. *Introduction to Grid Computing.* CRC Press, 2009. ISBN 9780429142536. doi: 10.1201/9781420074079.

[147] Mohammad Reza Mahmoudi, Mohammad Hossein Heydari, Sultan Noman Qasem, Amirhosein Mosavi, and Shahab S. Band. *Principal component analysis to study the relations between the spread rates of COVID-19 in high risks countries.* Alexandria Engineering Journal, 60(1):457–464, 2021. doi: 10.1016/j.aej.2020.09.013.

[148] Mateusz Majer, Jürgen Teich, Ali Ahmadinia, and Christophe Bobda. *The Erlangen slot machine: A dynamically reconfigurable FPGA-based computer.* Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 47(1): 15–31, 2007. ISSN 13875485. doi: 10.1007/s11265-006-0017-6.

[149] Maria Malik, Farnoud Farahmand, Paul Otto, Nima Akhlaghi, Tinoosh Mohsenin, Siddhartha Sikdar, and Houman Homayoun. *Architecture exploration for energy-efficient embedded vision applications: From general purpose processor to domain specific accelerator.* Computer Society Annual Symposium on VLSI (ISVLSI), pages 559–564. IEEE, 2016. doi: 10.1109/ISVLSI.2016.112.

[150] Yuyi Mao, Jun Zhang, and Khaled B. Letaief. *Dynamic Computation Offloading for Mobile-Edge Computing with Energy Harvesting Devices.* IEEE Journal on

Selected Areas in Communications, 34(12):3590–3605, 2016. ISSN 07338716. doi: 10.1109/JSAC.2016.2611964.

[151] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. *A Survey on Mobile Edge Computing: The Communication Perspective*. IEEE Communications Surveys and Tutorials, 19(4):2322–2358, 2017. ISSN 1553877X. doi: 10.1109/COMST.2017.2745201.

[152] Peter Marwedel. *Embedded Design System: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer, 2011. ISBN 0387292373.

[153] MATLAB. *HDL Coder - MATLAB & Simulink*, 2021, `https://www.mathworks.com/products/hdl-coder.html` (last visited: 2021-11-13).

[154] Mark D. McDonnell, Migel D. Tissera, Tony Vladusich, André Van Schaik, Jonathan Tapson, and Friedhelm Schwenker. *Fast, simple and accurate handwritten digit classification by training shallow neural network classifiers with the 'Extreme learning machine' algorithm*. PLoS ONE, 10(8):1–20, 2015. doi: 10.1371/journal.pone.0134254.

[155] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. *NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs*. ACM Transactions on Reconfigurable Technology and Systems, 11(3):1–24, 2017. doi: 10.1145/3284357.

[156] Changyun Miao and Chunqing Ye. *Research on IP net telephony system based on FPGA telephone terminal*. Conference on Environmental Science and Information Application Technology (ESIAT), volume 2, pages 415–417. IEEE, 2010. ISBN 9781424473885. doi: 10.1109/ESIAT.2010.5567329.

[157] Minghui Min, Liang Xiao, Ye Chen, Peng Cheng, Di Wu, and Weihua Zhuang. *Learning-Based Computation Offloading for IoT Devices with Energy Harvesting*. IEEE Transactions on Vehicular Technology, 68(2):1930–1941, 2019. arXiv ID: 1712.08768. ISSN 00189545. doi: 10.1109/TVT.2018.2890685.

[158] M. Mnif and C. Müller-Schloer. *Quantitative emergence*. Mountain Workshop on Adaptive and Learning Systems (SMCals), pages 78–84. IEEE, 2006. doi: 10.1109/SMCALS.2006.250695.

[159] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou,

Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. *Human-level control through deep reinforcement learning*. Nature, 518(7540): 529–533, 2015. ISSN 14764687. doi: 10.1038/nature14236.

[160] Volodymyr Mnih, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. *Asynchronous methods for deep reinforcement learning*. International Conference on Machine Learning (ICML), volume 4, pages 1928–1937. Association for Computing Machinery, 2016. ISBN 9781510829008. doi: 10.5555/3045390.3045594.

[161] M. H. Moattar and M. M. Homayounpour. *A simple but efficient real-time voice activity detection algorithm*. European Signal Processing Conference, pages 2549–2553. IOP Publishing Ltd, 2009. doi: 10.1088/1742-6596/705/1/012037.

[162] George B. Moody and Roger G. Mark. *The impact of the MIT-BIH arrhythmia database*. IEEE Engineering in Medicine and Biology Magazine, 2001. ISSN 07395175. doi: 10.1109/51.932724.

[163] Daniel C. Morais, Thiago Werlley B. Silva, Tiago P. Nascimento, Elmar Uwe Kurt Melcher, and Alisson V. Brito. *A distributed platform for integration of FPGA-based embedded systems*. Brazilian Symposium on Computing System Engineering (SBESC), pages 86–92. IEEE, 2017. ISBN 9781509026531. doi: 10.1109/SBESC.2016.021.

[164] Gero Mühl, Matthias Werner, Michael A. Jaeger, Klaus Herrmann, and Helge Parzyjegla. *On the Definitions of Self-Managing and Self-Organizing Systems*. Communication in Distributed Systems (KiVS), pages 1–11. VDE, 2007.

[165] Christian Müller-Schloer and Bernhard Sick. *Emergence in Organic Computing Systems: Discussion of a Controversial Concept*. International Conference on Autonomic and Trusted Computing (ATC), pages 1–16, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-38622-3.

[166] Christian Müller-Schloer and Sven Tomforde. *Organic Computing – Technical Systems for Survival in the Real World*, volume 578. Springer, 2017. ISBN 978-3-319-68476-5. doi: 10.1007/978-3-319-68477-2.

[167] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer. *Organic Computing - A Paradigm Shift for Complex Systems*. Springer, 2011. ISBN 9783034801294.

[168] Ranesh Kumar Naha, Saurabh Garg, Dimitrios Georgakopoulos, Prem Prakash Jayaraman, Longxiang Gao, Yong Xiang, and Rajiv Ranjan. *Fog computing: Survey of trends, architectures, requirements, and research directions*. IEEE Access, 6:47980–48009, 2018. ISSN 21693536. doi: 10.1109/ACCESS.2018.2866491.

[169] Hyunsuk Nam and Roman Lysecky. *Latency, power, and security optimization in distributed reconfigurable embedded systems.* International Parallel and Distributed Processing Symposium (IPDPS), pages 124–131. IEEE, 2016. ISBN 9781509021406. doi: 10.1109/IPDPSW.2016.40.

[170] Kevin Nam, Blair Fort, and Stephen Brown. *FISH: Linux system calls for FPGA accelerators.* International Conference on Field Programmable Logic and Applications (FPL), pages 1–4. IEEE, 2017. doi: 10.23919/FPL.2017.8056785.

[171] Tuan D.A. Nguyen and Akash Kumar. *PR-HMPSoC: A versatile partially reconfigurable heterogeneous Multiprocessor System-on-Chip for dynamic FPGA-based embedded systems.* International Conference on Field Programmable Logic and Applications (FPL), pages 1–6. IEEE, 2014. ISBN 9783000446450. doi: 10.1109/FPL.2014.6927492.

[172] NVIDIA. *CUDA Zone*, `https://developer.nvidia.com/cuda-zone` (last visited: 2021-11-07).

[173] NVIDIA. *Embedded Systems Developer Kits & Modules*, 2021, `https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/` (last visited: 2021-11-13).

[174] Jung Hwan Oh, Young Hyun Yoon, Ji Kwang Kim, Hyung Bin Ihm, Shin Hye Jeon, Tae Heon Kim, and Seung Eun Lee. *An FPGA-based Electronic Control Unit for Automotive Systems.* International Conference on Consumer Electronics (ICCE), pages 15–16. IEEE, 2019. ISBN 9781538679104. doi: 10.1109/ICCE.2019.8662003.

[175] Alexandru Corneliu Olteanu and Nicolae Tapus. *Offloading for mobile devices: A survey.* UPB Scientific Bulletin, Series C: Electrical Engineering, 76(1):3–16, 2014. ISSN 2286-3540.

[176] Hao Yi Ong, Kevin Chavez, and Augustus Hong. *Distributed Deep Q-Learning.* arXiv preprint, aug 2015. arXiv ID: 1508.04186, `http://arxiv.org/abs/1508.04186`.

[177] OpenCores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores [White List]*, 2002, `https://cdn.opencores.org/downloads/wbspec_b3.pdf` (last visited: 2021-12-23).

[178] Samet Oymak and Mahdi Soltanolkotabi. *Toward Moderate Overparameterization: Global Convergence Guarantees for Training Shallow Neural Networks.* IEEE Journal on Selected Areas in Information Theory, 1(1):84–105, 2020. doi: 10.1109/jsait.2020.2991332.

[179] Andrei Palade, Christian Cabrera, Gary White, Md Abdur Razzaque, and Siobhan Clarke. *Middleware for Internet of Things: A quantitative evaluation in small*

scale. International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), pages 1–6. IEEE, 2017. ISBN 9781538627228. doi: 10.1109/WoWMoM.2017.7974340.

[180] Andrei Palade, Christian Cabrera, Fan Li, Gary White, Mohammad Abdur Razzaque, and Siobhán Clarke. *Middleware for internet of things: an evaluation in a small-scale IoT environment*. Journal of Reliable Intelligent Environments, 4(1): 3–23, 2018. ISSN 2199-4668. doi: 10.1007/s40860-018-0055-4.

[181] Peichen Pan and Chih-Chang Lin. *A New Retiming-Based Technology Mapping Algorithm for LUT-Based FPGAs*. International Symposium on Field Programmable Gate Arrays (FPGA), FPGA '98, pages 35–42, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919785. doi: 10.1145/275107.275118.

[182] Angshuman Parashar, Michael Adler, Kermin E. Fleming, Michael Pellauer, and Joel S. Emer. *LEAP: A Virtual Platform Architecture for FPGAs*. Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL). unpublished, 2010.

[183] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. *Continual lifelong learning with neural networks: A review*. Neural Networks, 113:54–71, 2019. doi: 10.1016/j.neunet.2019.01.012.

[184] Donghyun Park, Seulgi Kim, Yelin An, and Jae Yoon Jung. *Lired: A light-weight real-time fault detection system for edge computing using LSTM recurrent neural networks*. Sensors, 18(7), 2018. ISSN 14248220. doi: 10.3390/s18072110.

[185] Song Jun Park, Dale R. Shires, and Brian J. Henz. *Coprocessor computing with FPGA and GPU*. Department of Defense High Performance Computing Modernization Program Users Group (DoD HPCMP), pages 366–370. IEEE, 2008. ISBN 9780769535159. doi: 10.1109/DoD.HPCMP.UGC.2008.69.

[186] Milan Patel, Yunchao Hu, Patrice Hédé, Jerome Joubert, Chris Thornton, Brian Naughton, Julian Roldan Ramos, Caroline Chan, Valerie Young, Soo Jin Tan, Daniel Lynch, Nurit Sprecher, Torsten Musiol, Carlos Manzanares, Uwe Rauschenbach, Sadayuki Abeta, Lan Chen, Kenji Shimizu, Adrian Neal, Peter Cosimini, Adam Pollard, and Guenter Klas. *Mobile-Edge Computing [White Paper]*, 2012, `https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v118-09-14.pdf` (last visited: 2021-12-23).

[187] Rushi Patel, Pierre Francois Wolfe, Robert Munafo, Mayank Varia, and Martin Herbordt. *Arithmetic and Boolean Secret Sharing MPC on FPGAs in the Data Center*. High Performance Extreme Computing Conference (HPEC). IEEE, 2020. ISBN 9781728192192. doi: 10.1109/HPEC43674.2020.9286159.

[188] Karl Pauwels, Matteo Tomasi, Javier Díaz Alonso, Eduardo Ros, and Marc M. Van Hulle. *A Comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features*. IEEE Transactions on Computers, 61(7): 999–1012, 2012. ISSN 00189340. doi: 10.1109/TC.2011.120.

[189] Karl Pearson. *LIII. On lines and planes of closest fit to systems of points in space*. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 2(11):559–572, 1901. ISSN 1941-5982. doi: 10.1080/14786440109462720.

[190] Max Planitz, William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. The Mathematical Gazette, 71(457):245, 1987. ISSN 00255572. doi: 10.2307/3616786.

[191] Chao Qian. *Energy Efficiency Analysis and Optimisation of Convolutional Neural Networks in Embedded FPGAs*. Master's thesis, University of Duisburg-Essen, 2019.

[192] Mohammad Rahimi, Rick Baer, Obimdinachi I. Iroezi, Juan C. Garcia, Jay Warrior, Deborah Estrin, and Mani Srivastava. *Cyclops: In situ image sensing and interpretation in wireless sensor networks*. International Conference on Embedded Networked Sensor Systems (SenSys), pages 192–204. Association for Computing Machinery, 2005. ISBN 159593054X. doi: 10.1145/1098918.1098939.

[193] Muhammad Zia Ur Rahman, Rafi Ahamed Shaik, and D. V.Rama Koti Reddy. *Efficient and simplified adaptive noise cancelers for ecg sensor based remote health monitoring*. IEEE Sensors Journal, 12(3):566–573, 2012. ISSN 1530437X. doi: 10.1109/JSEN.2011.2111453.

[194] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Cla. *Middleware for internet of things: A survey*. IEEE Internet of Things Journal, 3(1):70–95, 2016. ISSN 23274662. doi: 10.1109/JIOT.2015.2498900.

[195] Minzhen Ren. *Cordic-based Givens QR decomposition for MIMO detectors*. PhD thesis, Georgia Institute of Technology, 2013.

[196] Xavier Revés, Vuk Marojevic, Ramon Ferrús, and Antoni Gelonch. *FPGA's middleware for software defined radio applications*. International Conference on Field Programmable Logic and Applications (FPL), pages 598–601. IEEE, 2005. ISBN 0780393627. doi: 10.1109/FPL.2005.1515794.

[197] Bernhard Rinner and Wayne Wolf. *An introduction to distributed smart cameras*. Proceedings of the IEEE, 96(10):1565–1575, 2008. ISSN 00189219. doi: 10.1109/JPROC.2008.928742.

[198] Bernhard Rinner, Lukas Esterle, Jennifer Simonjan, Georg Nebehay, Roman Pflugfelder, Gustavo Fernandez Dominguez, and Peter R. Lewis. *Self-Aware and*

Self-Expressive camera networks. Computer, 48(7):21–28, 2015. ISSN 00189162. doi: 10.1109/MC.2015.209.

[199] Alfonso Rodriguez, Juan Valverde, Cesar Castanares, Jorge Portilla, Eduardo De La Torre, and Teresa Riesgo. *Execution modeling in self-Aware FPGA-based architectures for efficient resource management*. International Symposium on Reconfigurable and Communication-centric Systems-on-Chip (ReCoSoC). IEEE, 2015. ISBN 9781467379427. doi: 10.1109/ReCoSoC.2015.7238086.

[200] Alfonso Rodríguez, Juan Valverde, Jorge Portilla, Andrés Otero, Teresa Riesgo, and Eduardo De La Torre. *FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo3 framework*. Sensors, 18(6), 2018. ISSN 14248220. doi: 10.3390/s18061877.

[201] Juan J. Rodriguez-Andina, Maria J. Moure, and Maria D. Valdes. *Features, design tools, and application domains of FPGAs*. IEEE Transactions on Industrial Electronics, 54(4):1810–1823, 2007. ISSN 02780046. doi: 10.1109/TIE.2007.898279.

[202] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. *Mobile edge computing, Fog et al.: A survey and analysis of security threats and challenges*. Future Generation Computer Systems, 78(2):680–698, 2018. ISSN 0167739X. doi: 10.1016/j.future.2016.11.009.

[203] Karen Rose, Scott Eldridge, and Lyman Chapin. *An Overview of Internet of Things: Understanding the Issues and Challenges of a More Connected World*, 2015. ISSN 1536-5026, `https://www.internetsociety.org/wp-content/uploads/2017/08/ISOC-IoT-Overview-20151221-en.pdf` (last visited: 2021-12-23).

[204] Rashmi Ranjan Rout and Soumya K. Ghosh. *Enhancement of lifetime using duty cycle and network coding in wireless sensor networks*. IEEE Transactions on Wireless Communications, 12(2):656–667, 2013. ISSN 15361276. doi: 10.1109/TWC.2012.111412.112124.

[205] G Rummery and Mahesan Niranjan. *On-Line Q-Learning Using Connectionist Systems*. Technical Report CUED/F-INFENG/TR 166, 1994.

[206] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, 4th edition, 2002. ISBN 9781292153964.

[207] Mahsa Salmani, Foad Sohrabi, Timothy N. Davidson, and Wei Yu. *Multiple Access Binary Computation Offloading via Reinforcement Learning*. Canadian Workshop on Information Theory (CWIT). IEEE, 2019. ISBN 9781728109541. doi: 10.1109/CWIT.2019.8929930.

[208] M. D. Santambrogio, V. Rana, I. Beretta, and D. Sciuto. *Operating system runtime management of partially dynamically reconfigurable embedded systems.* Workshop on Embedded Systems for Real-Time Multimedia, pages 1–10. IEEE, oct 2010. ISBN 978-1-4244-9084-4. doi: 10.1109/ESTMED.2010.5666975.

[209] Bernd Dominik Schaefer. *Elastic Computation Placement in Edge-based Environments.* PhD thesis, University of Mannheim, 2019.

[210] Dominik Schäfer, Janick Edinger, Justin Mazzola Paluska, Sebastian Vansyckel, and Christian Becker. *Tasklets: "better than Best-Effort" computing.* International Conference on Computer Communications and Networks (ICCCN). IEEE, 2016. ISBN 9781509022793. doi: 10.1109/ICCCN.2016.7568580.

[211] Gregor Schiele, Alwyn Burger, and Christopher Cichiwskyj. *The Elastic Node: An Experimentation Platform for Hardware Accelerator Research in the Internet of Things.* International Conference on Autonomic Computing (ICAC), pages 84–94. IEEE, 2019. ISBN 9781728124117. doi: 10.1109/ICAC.2019.00020.

[212] Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. *Adaptivity and self-organization in organic computing systems.* ACM Transactions on Autonomous and Adaptive Systems, 5(3):1–32, 2010. ISSN 15564665. doi: 10.1145/1837909.1837911.

[213] Philip Schmidt. *Hardware-/Software-Codesign for an Embedded Energy Monitoring Daughterboard.* Bachelor's thesis, University of Duisburg-Essen, 2019.

[214] Uwe Schwiegelshohn, Rosa M. Badia, Marian Bubak, Marco Danelutto, Schahram Dustdar, Fabrizio Gagliardi, Alfred Geiger, Ladislav Hluchy, Dieter Kranzlmller, Erwin Laure, Thierry Priol, Alexander Reinefeld, Michael Resch, Andreas Reuter, Otto Rienhoff, Thomas Rter, Peter Sloot, Domenico Talia, Klaus Ullmann, Ramin Yahyapour, and Gabriele Von Voigt. *Perspectives on grid computing.* Future Generation Computer Systems, 26(8):1104–1115, oct 2010. doi: 10.1016/J.FUTURE.2010.05.010.

[215] Khurram Shahzad. *Energy Efficient Wireless Sensor Node Architecture for Data and Computation Intensive Applications.* Doctoral thesis, Mid Sweden University, 2014.

[216] Khurram Shahzad and Bengt Oelmann. *Investigating Energy Consumption of an SRAM-based FPGA for Duty- Cycle Applications.* Advances in Parallel Computing, 25:548–559, 2014. doi: 10.3233/978-1-61499-381-0-548.

[217] Khurram Shahzad, Peng Cheng, and Bengt Oelmann. *SENTIOF : An FPGA Based High-Performance and Low-Power Wireless Embedded Platform.* Federal Conference on Computer Science and Information Systems (FedCSIS), pages 901–906. IEEE, 2013. ISBN 978-83-60810-52-1.

[218] Ran Shu, Peng Cheng, Guo Chen, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. *Direct Universal Access: Making Data Center Resources Available to FPGA*. USENIX Conference on Networked Systems Design and Implemented (NSDI), pages 127–140. Association for Computing Machinery, 2019. doi: 10.5555/3323234.3323246.

[219] Thiago W.B. Silva, Daniel C. Morais, Halamo G.R. Andrade, Antonio M.N. Lima, Elmar U.K. Melcher, and Alisson V. Brito. *Environment for integration of distributed heterogeneous computing systems*. Journal of Internet Services and Applications, 9(1):1–17, 2018. doi: 10.1186/s13174-017-0072-1.

[220] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. *Resource provisioning for IoT services in the fog*. International Conference on Service-Oriented Computing and Applications (SOCA), pages 32–39. IEEE, 2016. ISBN 9781509047819. doi: 10.1109/SOCA.2016.10.

[221] Angela C Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. *Parallelism via multithreaded and multicore CPUs*. Computer, 43(3): 24–32, 2010. ISSN 00189162. doi: 10.1109/MC.2010.75.

[222] Lukas Sommer, Jens Korinth, and Andreas Koch. *OpenMP device offloading to FPGA accelerators*. International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 201–205. IEEE, 2017. doi: 10.1109/ASAP.2017.7995280.

[223] Haoran Song. *The Application of Computer Vision in Responding to the Emergencies of Autonomous Driving*. Proceedings - 2020 International Conference on Computer Vision, Image and Deep Learning, CVIDL 2020, (Cvidl):1–5, 2020. doi: 10.1109/CVIDL51233.2020.00008.

[224] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. *In-Situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems*. International Symposium on High Performance Computer Architecture (HPCA), pages 92–103. IEEE, 2018. doi: 10.1109/HPCA.2018.00018, `http://ieeexplore.ieee.org/document/8327001/`.

[225] C. O. S. Sorzano, J. Vargas, and A. Pascual Montano. *A survey of dimensionality reduction techniques*. arXiv preprint, pages 1–35, 2012. arXiv ID: 1403.2877.

[226] Stacy Wegner, Al Cowsky, Chad Davis, Dick James, Daniel Yang, Ray Fontaine, and Jim Morrison. *Apple iPhone 7 Teardown*, 2016, `https://www.techinsights.com/blog/apple-iphone-7-teardown` (last visited: 2020-11-12).

[227] Thomas J. Sullivan, Stephen R. Deiss, and Gert Cauwenberghs. *A low-noise, non-contact EEG/ECG sensor*. Biomedical Circuits and Systems Conference Health-

care Technology (BiOCAS), pages 154–157. IEEE, 2007. ISBN 142441525X. doi: 10.1109/BIOCAS.2007.4463332.

[228] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, An Introduction.* Number 1. The MIT Press, second edition, 2020. ISBN 978-0262193986.

[229] Andrew Tanenbaum and David J. Wetherall. *Computer Networks.* Prentice Hall, 5th edition, 2010.

[230] Andrew S Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* Pearson, 2nd edition, 2006. ISBN 978-0-13-239227-3.

[231] Qinqin Tang, Renchao Xie, Fei Richard Yu, Tao Huang, and Yunjie Liu. *Decentralized computation offloading in IoT fog computing system with energy harvesting: A dec-POMDP approach.* IEEE Internet of Things Journal, 7(6):4898–4911, 2020. ISSN 23274662. doi: 10.1109/JIOT.2020.2971323.

[232] David Tennenhouse. *Proactive computing.* Communications of the ACM, 43(5): 43–50, 2000. ISSN 00010782. doi: 10.1145/332833.332837.

[233] Gerald Tesauro. *Practical issues in temporal difference learning.* Machine Learning, 8(3):257–277, 1992. ISSN 0885-6125. doi: 10.1007/bf00992697.

[234] Eiji Tokunaga, Masahiro Nemoto, and Tatsuo Nakajima. *Object-Oriented Middleware Infrastructure for Distributed Augmented Reality.* International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), page 156. IEEE, 2003. ISBN 0769519288. doi: 10.5555/850985.855556.

[235] J. Tomarakos, C. Duggan, and S. Steyerl. *32-Bit SIMD SHARC architecture digital audio signal processing applications.* Journal of the Audio Engineering Society, 48 (3):220–229, mar 2000. ISSN 00047554.

[236] Sven Tomforde and Martin Goller. *To adapt or not to adapt: A quantification technique for measuring an expected degree of self-adaptation.* Computers, 9(1), 2020. ISSN 2073431X. doi: 10.3390/computers9010021.

[237] Sven Tomforde, Andreas Brameshuber, Jörg Hahner, and Christian Müller-Schloer. *Restricted on-line learning in real-world systems.* Congress of Evolutionary Computation (CEC), pages 1628–1635. IEEE, 2011. doi: 10.1109/CEC.2011.5949810.

[238] Sven Tomforde, Holger Prothmann, Jürgen Branke, Jörg Hähner, Moez Mnif, Christian Müller-Schloer, Urban Richter, and Hartmut Schmeck. Observation and Control of Organic Systems. Organic Computing - A Paradigm Shift for Complex Systems, pages 325–338. Springer, 2011. doi: 10.1007/978-3-0348-0130-0_21.

[239] Sven Tomforde, Jan Kantert, and Bernhard Sick. *Measuring self-organisation at runtime a quantification method based on divergence measures.* International Conference on Agents and Artificial Intelligence (ICAART), volume 2, pages 96–106. SciTePress, 2017. ISBN 9789897582196. doi: 10.5220/0006240400960106.

[240] Sven Tomforde, Bernhard Sick, and Christian Müller-Schloer. *Organic Computing in the Spotlight.* arXiv preprint, 2017. arXiv ID: 1701.08125.

[241] Patrick Urban. *Implementation of Distributed Computer Vision using Embedded FPGAs.* Master's thesis, University of Duisburg-Essen, 2020.

[242] Ryan J. Urbanowicz and Jason H. Moore. *Learning Classifier Systems: A Complete Introduction, Review, and Roadmap.* Journal of Artificial Evolution and Applications, pages 1–25, 2009. ISSN 1687-6229. doi: 10.1155/2009/736398.

[243] Miroslav Valov. *Over-the-air Updating of an Embedded Heterogeneous Platform Using 802.15.4.* Bachelor's thesis, University of Duisburg-Essen, 2019.

[244] Juan Valverde, Andres Otero, Miguel Lopez, Jorge Portilla, Eduardo de la Torre, and Teresa Riesgo. *Using SRAM based FPGAs for power-aware high performance wireless sensor networks.* Sensors, 12(3):2667–2692, 2012. doi: 10.3390/s120302667.

[245] Jesper E. van Engelen and Holger H. Hoos. *A survey on semi-supervised learning.* Machine Learning, 109(2):373–440, 2020. ISSN 15730565. doi: 10.1007/s10994-019-05855-6.

[246] Ivan Vido, Ivana Škorić, Dominik Mitrović, Milena Milošević, and Marijan Herceg. *Automotive Vision Grabber: FPGA design, cameras and data transfer over PCIe.* Zooming Innovation in Consumer Technologies Conference (ZINC), pages 103–108. IEEE, 2019. doi: 10.1109/ZINC.2019.8769348.

[247] Félix Jesús Villanueva, David Villa, Francisco D. Moya, Jesús Barba, Fernando Rincón, Juan C. López, and Jesús Barba Romero. *Lightweight Middleware for Seamless HW-SW Interoperability, with Application to Wireless Sensor Network.* Design, Automation & Test in Europe Conference & Exhibition (DATE), volume 67, pages 1042–1047. IEEE, 2007. doi: 10.1109/DATE.2007.364431.

[248] Steve Vinoski. *CORBA: Integrating diverse applications within distributed heterogeneous environments.* IEEE Communications Magazine, 35(2):46–55, 1997. doi: 10.1109/35.565655.

[249] Jack E. Volder. *Birth of CORDIC.* Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 25(2):101–105, 2000. ISSN 09225773. doi: 10.1023/A:1008110704586.

[250] Jack E. Volder. The CORDIC trigonometric computing technique. Computer Arithmetic: Volume I, pages 245–249. World Scientific, 2015. doi: 10.1142/9789814651578.

[251] Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. *Peer-to-peer computing: Principles and applications*. Springer, 2010. doi: 10.1007/978-3-642-03514-2.

[252] Manu Vyas. *Trends of FPGA use in automotive engineering*. International Conference on Recent Trends in Electronics, Information and Communication Technology (RTEICT), number 2004, pages 580–591. IEEE, 2018. ISBN 9781538624401. doi: 10.1109/RTEICT42901.2018.9012495.

[253] Zhiyong Wang, Longxi Chen, Lifeng Wang, and Guangqiang Diao. *Recognition of Audio Depression Based on Convolutional Neural Network and Generative Antagonism Network Model*. IEEE Access, 8:101181–101191, 2020. ISSN 21693536. doi: 10.1109/ACCESS.2020.2998532.

[254] Christopher Watkins. *Learning from delayed rewards*. Ph.d. thesis, King's College, 1989.

[255] Christopher J.C.H. Watkins and Peter Dayan. *Technical Note: Q-Learning*. Machine Learning, 8(3):279–292, 1992. doi: 10.1023/A:1022676722315.

[256] Joel L. Wilder, Vladimir Uzelac, Aleksandar Milenković, and Emil Jovanov. *Run-time hardware reconfiguration in wireless sensor networks*. Annual Southeastern Symposium on System Theory (SSST), pages 154–158. IEEE, 2008. doi: 10.1109/SSST.2008.4480210.

[257] Barry Wilkinson. Introduction to Grid Computing. Grid Computing, pages 23–56. CRC Press, 2020. doi: 10.1201/9781420069549-6.

[258] John A. Williams and Neil W. Bergmann. *Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip*. International Conference On Engineering of Reconfigurable Systems and Algorithms (ERSA), pages 163–169. CSREA Press, 2004. ISBN 1-932415-42-4.

[259] Stewart W. Wilson. *Classifier Fitness Based on Accuracy*. Evolutionary Computation, 3(2):149–175, 1995. ISSN 1063-6560. doi: 10.1162/evco.1995.3.2.149.

[260] Philipp Winnekens. *Development of an Interface Description Language and Stub/Skeleton Generator for Embedded Heterogeneous Multicore Systems*. Master's thesis, University of Duisburg-Essen, 2020.

[261] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. *A Comprehensive Survey on Graph Neural Networks*. IEEE Transactions on Neural Networks and Learning Systems, 32(1):4–24, 2021. arXiv ID: 1901.00596. ISSN 21622388. doi: 10.1109/TNNLS.2020.2978386.

[262] Xilinx. *7 Series FPGAs Configuration (UG470) [White Paper]*, https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf (last visited: 2021-12-23).

[263] Xilinx. *AXI Reference Guide (UG761) [White Paper]*, 2011, https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf (last visited: 2021-12-23).

[264] Xilinx. *EDK Concepts, Tools, and Techniques - A Hands-On Guide to Effective Embedded System Design (UG683)*, 2013, https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/edk_ctt.pdf (last visited: 2021-12-23).

[265] Xilinx. *7 Series FPGAs Configurable Logic Block (UG474) [White Paper]*, 2014, https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf (last visited: 2021-12-23).

[266] Xilinx. *7 Series FPGAs Datasheet (DS180) [White Paper]*, 2020, https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (last visited: 2021-12-23).

[267] Xilinx. *Platform Studio and the Embedded Development Kit (EDK)*, 2021, https://www.xilinx.com/products/design-tools/platform.html (last visited: 2021-10-10).

[268] Xilinx. *Zynq-7000 SoC*, 2021, https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (last visited: 2021-11-13).

[269] Xilinx Inc. *Vivado High-Level Synthesis*, 2018, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html (last visited: 2021-12-23).

[270] Xilinx Inc. *7 Series FPGAs Memory Resources: User Guide (UG473) [White Paper]*, 2019, https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (last visited: 2021-12-23).

[271] Zhiyuan Xu, Yanzhi Wang, Jian Tang, Jing Wang, and Mustafa Cenk Gursoy. *A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs.* International Conference on Communications (ICC), pages 1–6. IEEE, 2017. ISBN 9781467389990. doi: 10.1109/ICC.2017.7997286.

[272] Tianyu Yang, Yulin Hu, M. Cenk Gursoy, Anke Schmeink, and Rudolf Mathar. *Deep reinforcement learning based resource allocation in low latency edge computing networks.* International Symposium on Wireless Communication Systems (ISWCS). IEEE, oct 2018. ISBN 9781538650059. doi: 10.1109/ISWCS.2018.8491089.

[273] Xiaoling Yang, Baohua Tan, Jiehua Ding, Jinye Zhang, and Jiaoli Gong. *Comparative study on voice activity detection algorithm.* International Conference on Electrical and Control Engineering (ICECE), pages 599–602. IEEE, 2010. ISBN 9780769540313. doi: 10.1109/iCECE.2010.153.

[274] Shanhe Yi, Cheng Li, and Qun Li. *A Survey of Fog Computing: Concepts, Applications and Issues.* Workshop on Mobile Big Data (Mobidata), 2015. ISBN 9781450335249. doi: 10.1145/2757384.2757397.

[275] Shanhe Yi, Zhengrui Qin, and Qun Li. *Security and privacy issues of fog computing: A survey.* International Conference on Wireless Algorithms, Systems, and Applications (WASA). Springer, 2015. doi: 10.1007/978-3-319-21837-3_67.

[276] Xianghe Yi and Jian-jun Tan. *Enterprise Human Resource Management System Based on Fpga and Machine Learning.* Solid State Technology, 63(4):7546–7555, 2020.

[277] Tjalling J. Ypma. *Historical development of the Newton-Raphson method.* SIAM Review, 37(4):531–551, 1995. ISSN 00361445. doi: 10.1137/1037125.

[278] Xiaoyu Yu, Jianlin Gao, Yuwei Wang, Jie Miao, Ephrem Wu, Heng Zhang, Yu Meng, Bo Zhang, Biao Min, and Dewei Chen. *A data-center FPGA acceleration platform for convolutional neural networks.* International Conference on Field-Programmable Logic and Applications (FPL), pages 151–158. IEEE, 2019. ISBN 9781728148847. doi: 10.1109/FPL.2019.00032.

[279] Johannes Zeppenfeld, Abdelmajid Bouajila, Walter Stechele, Andreas Bernauer, Oliver Bringmann, Wolfgang Rosenstiel, and Andreas Herkersdorf. Applying ASoC to Multi-core Applications for Workload Management. Organic Computing - A Paradigm Shift for Complex Systems, pages 461–472. Springer, 2011. doi: 10.1007/978-3-0348-0130-0_30.

[280] Yang Zhang, Dusit Niyato, Ping Wang, and Chen Khong Tham. *Dynamic offloading algorithm in intermittently connected mobile cloudlet systems.* International Conference on Communications (ICC), pages 4190–4195. IEEE, 2014. ISBN 9781479920037. doi: 10.1109/ICC.2014.6883978.

[281] Huimin Zhao, Jianjie Zheng, Junjie Xu, and Wu Deng. *Fault diagnosis method based on principal component analysis and broad learning system.* IEEE Access, 7:99263–99272, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2929094.

DuEPublico

Duisburg-Essen Publications online

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

ub | universitäts bibliothek

Diese Dissertation wird via DuEPublico, dem Dokumenten- und Publikationsserver der Universität Duisburg-Essen, zur Verfügung gestellt und liegt auch als Print-Version vor.

DOI: 10.17185/duepublico/78189
URN: urn:nbn:de:hbz:465-20230517-114021-3

Alle Rechte vorbehalten.