Asynchronous Programming Using C++ Coroutines in Embedded and Edge Computing

College of Science and Engineering

Thesis submitted by

Bruce Belson, BA (Hons) MA

in July 2024

for the degree of

Doctor of Philosophy

Acknowledgements

I am grateful to the following for their help and support through the long and winding road that led to the completion of this thesis.

To my primary advisor, Associate Professor Bronson Philippa, thank you for your extraordinary patience and wisdom, offered over a long span of years, which guided me along this fascinating and enjoyable journey.

To the members of my advisory committee, Dr Jason Holdsworth and Professor Wei Xiang, I offer my thanks for your support and guidance during the development of the ideas and conclusions of the research chapters, and for your invaluable help in improving their final products.

To my colleagues in the JCU Cairns Engineering department, thank you for your continued moral support and good cheer and for your patience when explaining what must have so often been – for you – the transparently obvious.

To my three most important supporters, who patiently listened to increasingly abstruse problems and explanations, and did not flinch from providing criticism when it was needed: thank you, Anabel, Grace and Rose.

To my late parents, Dr William Belson and Dr Margaret Harris Belson, who alongside all their other gifts, patiently and steadfastly attempted to encourage rationalism and a sense of fairness in their children, I dedicate this work.

Statement of the Contribution of Others

I gratefully acknowledge the following contributions towards this work.

Financial support

Financial support was provided by The Australian Government Research Training Program (RTP) Scholarship. James Cook University and the JCU College of Science and Engineering provided additional support through annual funding and through supplementary grants for equipment.

Editorial support

Editorial support for the entire thesis was provided by my primary supervisor, Associate Professor Bronson Philippa of James Cook University.

Advisory Committee

Associate Professor Bronson Philippa, James Cook University
Dr Jason Holdsworth, James Cook University
Professor Wei Xiang, James Cook University & La Trobe University

Technical support

Technical support was provided by Ben Lyons, Engineering, James Cook University.

Contributions to co-authored publications

Contributions to co-authored publications that form chapters 2-5 of this thesis were made as below, categorised as Contributor Roles Taxonomy (CRediT) Roles. I am the lead author for each of the co-authored publications.

Chapter	Co-author	Conceptualization	Methodology	Software	Validation	Investigation	Data Curation	Writing - Original Draft	Writing - Review & Editing	Visualization	Supervision
2	Bruce Belson		✓	✓	✓	✓	✓	✓	✓	✓	
	Jason Holdsworth								✓	✓	
	Wei Xiang								✓		
	Bronson Philippa	✓	✓			✓			✓	✓	✓
3	Bruce Belson	✓	✓	✓		✓		✓	✓	✓	
	Jason Holdsworth								✓		
	Wei Xiang								✓		
	Bronson Philippa	✓	✓						✓	✓	✓
4	Bruce Belson	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	Bronson Philippa	✓	✓						✓	✓	✓
5	Bruce Belson	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	Jason Holdsworth								✓	✓	
	Bronson Philippa	✓	✓						✓	✓	✓

Statement of Access

I, the undersigned, author of this work, understand that James Cook University will make this thesis available for use within the University Library and, via the Australian Digital Thesis network, for use elsewhere.

I understand that, as an unpublished work, a thesis has significant protection under the Copyright Act and;

I do not wish to place any further restrictions on access to this work. However, any use of its content should be acknowledged and may be restricted by future patents.

Name: _____ Signature: _____ Date:_____

Abstract

As the Internet of Things (IoT) continues to expand in both size and economic importance, the development tools used for embedded and edge computing must evolve to address the increasing demands of these resource-constrained platforms. One critical challenge in this domain is the effective management of asynchronous and concurrent execution, which remains a time-consuming and error-prone aspect of development. Addressing this challenge is crucial, as it directly impacts the reliability, maintainability, and performance of embedded systems.

This thesis investigates the potential of coroutines, a feature introduced in the 2020 C++ standard, to improve the ease of use, safety, and efficiency of asynchronous and concurrent programming for embedded systems. Coroutines are of particular interest because they offer a structured and lightweight approach to managing concurrency compared to traditional methods. Furthermore, as a language-native feature, they provide a mainstream solution which will benefit from continued compiler optimisations and increasing community support. While the coroutine specification contained in the C++ 20 standard has been increasingly well supported for general-purpose systems, its applicability to resource-constrained embedded platforms has been underexplored. This work aims to fill that gap through a systematic survey, technical analysis, experimental library development, benchmarking, and a real-world case study.

Significant research gaps were identified in the study of tools for concurrent and asynchronous programming on the resource-constrained platforms that predominate in embedded and edge computing. The implementation of coroutines required by the C++ 2020 standard was examined in detail. An implementation of the standard was built specifically for resource-constrained devices without operating systems and without relying on the C++ Standard Template Library (STL). Performance benchmarks demonstrated that coroutines provide notable advantages over industry-standard real-time operating systems, achieving up to 12x faster execution speeds and a code size reduction of up to 6x.

Next, coroutines were applied to machine learning inference algorithms executed on edge de-

vices. By enhancing memory access patterns and reducing CPU cache misses through coroutine-based task suspension and resumption, speed improvements of 8–60% were achieved without modifying the underlying algorithms. These transformations required minimal code changes, underscoring the practicality and efficiency of coroutine integration.

A case study further validated the performance and usability of coroutines in real-world applications. Deeply iterative and resource-intensive tasks were reorganized into coroutine-based sub-tasks whose suspension and resumption were managed by a custom scheduler. Substantial gains in both performance and energy efficiency were observed, including execution time reductions of up to 20.5% and energy consumption reductions of up to 20%. Peak power usage reductions of up to 4.5% were observed, while peak current was reduced by up to 25 mA. These peak power reductions indicate a potential for extending battery life in rechargeable devices—a critical consideration for many IoT applications. While the transformations applied to the source code were simple and generic, the performance improvements in speed, energy consumption and peak power levels were significant.

The findings of this thesis demonstrate that C++ coroutines offer significant advantages for embedded and edge systems, both in terms of performance and energy efficiency. Although introducing coroutine-based solutions requires some development effort, the benefits in memory efficiency, execution speed, and energy consumption make them a compelling choice, and the development costs are predictable and manageable. These results also provide a strong rationale for adopting C++ in domains where C currently dominates, given the advantages of language-native coroutine support.

List of Publications

The following publications were produced during the candidature, and are included as thesis chapters as described below.

Chapter	Details of publication	Status
2	B. Belson, J. Holdsworth, W. Xiang and B. Philippa, "A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems" in *ACM Transactions on Embedded Computing Systems,* Volume 18, Issue 3, May 2019, Article No.: 21, pp 1–21, doi: 10.1145/3319618.	Published
3	B. Belson, W. Xiang, J. Holdsworth and B. Philippa, "C++20 Coroutines on Microcontrollers—What We Learned," in *IEEE Embedded Systems Letters*, vol. 13, no. 1, pp. 9-12, March 2021, doi: 10.1109/LES.2020.2973397.	Published
4	B. Belson and B. Philippa, "Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines", in *Proceedings of the 2022 IEEE 25th International Conference on Computational Science and Engineering (CSE)*, doi: 10.1109/CSE57773.2022.00011.	Published
5	B. Belson, J. Holdsworth and B. Philippa, "Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines", submitted to *Elsevier Internet of Things*, 28-Apr-2024	Submitted

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACM	Association for Computing Machinery
ADC	Analog-to-Digital Converter
API	Application Programming Interface
BPFI	Ball Passing Frequency Inner race
BPFO	Ball Passing Frequency Outer race
CNN	Convolutional Neural Network
CPS	Continuation-passing Style
CPU	Central Processing Unit
CSF	Coroutine Stack Frame
DOI	Digital Object Identifier
DSRM	Design Science Research Methodology
FFT	Fast Fourier Transform
FSM	Finite State Machine
GPIO	General-Purpose Input/Output
I^2C	Inter-Integrated Circuit (serial communication bus)
IEEE	Institute of Electrical and Electronics Engineers
IMU	Inertial measurement unit
IoT	Internet of Things
ISBN	International Standard Book Number
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
LU	Lower-Upper
ML	Machine Learning
N4680	ISO/IEC Proposed Draft Technical Specification - C++ Extensions for Coroutines

PHM	Prognostic Health Management
RAM	Random-Access Memory
ROM	Read-Only Memory
RTOS	Real-Time Operating System
SLOC	Source Lines Of Code
STL	Standard Template Library (C++)
SVM	Support Vector Machine
UDP	User Datagram Protocol
WCET	Worst-case execution time
WSAN	Wireless Sensor and Actuator Network
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Background

1.1.1 Embedded systems and the Internet of Things

Embedded systems [106, 194] and the Internet of Things (IoT) [2, 11, 70] often require computing systems whose run-time life-cycle exists, in the main, without direct human oversight or interference. The impact of the IoT continues to grow, in both economic [118, 147] and social [164] terms, as does the number of developed applications [117, 122] and the volume of data [14, 192] gathered.

This growth in both the importance and the size of the IoT requires that closer attention be paid to the software engineering of the resource-constrained embedded systems that are frequently deployed as part of the IoT, particularly as it impacts on their security [148, 166], reliability [70] and privacy [193]. These resource-constrained platforms – often referred to as 'edge devices' – may possess significantly less processing power, storage and power supply than mainstream computing devices, and consequently present very different software challenges.

1.1.2 Asynchronous programming

The architecture of many embedded systems is event-driven: the system holds multiple tasks in a waiting state, until an external event occurs. This architecture requires asynchronous code, which is often challenging to write because the code is split into phases, for event initiation and for the 'event handler' that manages the response to the event [62, 107, 121]. The addition of further asynchronous sub-tasks into the system as part

of the response to events adds to code complexity and the split-phase design causes a gap to grow between the intent of the system and its implementation as source code, reducing the readability and maintainability of the system [24, 52, 116, 88].

Furthermore, the requirement that the various tasks within a system be managed concurrently can introduced dangerous instability. Unexpected behaviour can be caused by race conditions [129] and locks: it is thus all the more critical that source code be simple and intelligible.

A common model for the management of split-phase programming is the finite state machine (FSM) [105], which brings its own problems of transparency and complexity, particularly with regard to composition [35]. Another approach to simplifying the problems posed by split-phase programming is the use of continuation passing style, whereby one function is called and is passed the code entry point to invoke after its completion. While this allows connected actions to be presented in the same code location, the use of this style in Javascript has famously led to the ”pyramid of doom” or ”callback hell” phenomenon [24, 52, 116, 88] when multiple sequential operations are composed.

The ‘async/await’ pattern [20, 136, 182] is a more elegant solution to the problem of split-phase coding. The pattern allows a function to pause its execution until a particular asynchronous operation completes. Importantly, it allows the use of a direct programming style, in which the various phases can be written in a natural order, with the use of conditions, loops and other standard control flows. The pattern has been used successfully in several languages, including C# [20, 136], JavaScript [51], Python [157, 185], Swift [149, 90], Rust [120, 197] and C++ [80].

1.1.3 Coroutines

The execution of a conventional subroutine begins at a single entry point, and exit occurs only once, after which the subroutine is considered complete. The subroutine does not maintain execution state between invocations, so when a subroutine that has been exited is invoked again, it starts from the beginning. This is not a suitable limitation for a native implementation of the ‘async/await’ pattern.

Coroutines differ from subroutines in that they allow multiple entry points for suspending and resuming execution at certain locations [36, 96, 119]. Unlike traditional

subroutines, coroutines enable cooperative multitasking by pausing and resuming functions at designated points. This capability makes them particularly suitable for implementing the 'async/await' pattern (as well as other control problems such as generators or producer/consumer patterns). The 'await' keyword instructs the compiler to jump out of the currently executing coroutine immediately after the statement marked with 'await', thereby suspending its execution. When the coroutine is resumed it restarts from the statement immediately following the 'await' statement. The 'async' keyword can be used to indicate that a function contains one or more 'await' statements and should therefore be executed as a coroutine.

Since the first appearance of coroutines in Simula in the 1960s [38], coroutines have been implemented – in various forms – across different programming languages, including C# [20, 136], Python [157, 185], JavaScript [51], and Kotlin [54].

A native implementation of coroutines requires that the state of local variables within the coroutine and of the progress of execution be maintained between invocations: improved programming ergonomics depends on the presence of these characteristics. The location in memory of the state information – as well as the degree of control that the developer maintains over that location – is a critical issue for embedded programmers [171].

1.1.4 The C programming language

The C language was first developed by Ritchie and others at Bell Labs between 1969 and 1973, publicly defined by Kernighan and Ritchie in 1978 and standardised by the ANSI X3J11 committee from 1983 [153]. It has long been associated with embedded software including operating system kernels – its original use was as a system implementation language for the Unix kernel for the PDP-11 [153]. It continues to be the dominant language in embedded systems development; while the percentage of developers reporting that they used C as the main language for embedded projects has fallen gradually since 2015 (see Fig. 1.1), C was still the dominant language reported in the 2023 AspenCore Embedded Markets Study, with 52% of respondents using C in their current project [190, 189, 9, 10, 37].

Because the C language can create very compact code, with minimal runtime overhead, and because the language is capable of directly addressing hardware registers

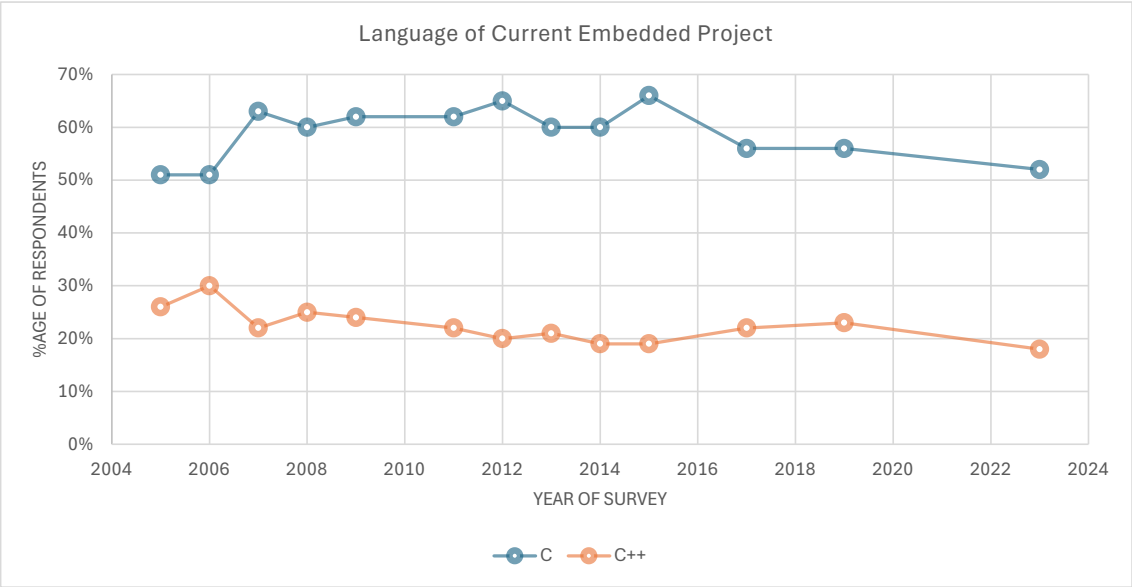


Figure 1.1: Use of C and C++ as language of current embedded development project, according to a survey completed by a readership of embedded developers, in answer to the question 'My *current* embedded project is programmed mostly in:'. Data drawn from [190, 189, 9, 10, 37].

using pointers [153], C is well suited to the task of running embedded software. However, the management of memory in C – and the use of pointers in particular – has led to a history of security vulnerabilities, including buffer overflows and underflows [167]. These and other vulnerabilities have been addressed by the introduction of coding standards such as MISRA [13], which restrict the programmer to a subset of the C language, and by the use of static analysis tools [174], which endeavour to find errors in source code.

1.1.5 The C++ programming language

The C++ language was designed by Bjarne Stroustrup as 'C with classes' and first used in 1980, before being made commercially available in 1985, and standardised between 1990 and 1998 [176, 177]. The language aimed to be a 'better C' that would 'make programming more enjoyable', as well as supporting data abstraction, object-oriented programming and generic programming [177].

Fig. 1.2 illustrates the gap between the intention to use C++ for embedded projects and the actual use. Based on data collected for the Embedded Markets Study between 2005 and 2023 [190, 189, 9, 10, 37], the figure compares the proportion of developers using C++ for their current development project with the proportion who said they

planned to use C++ for their next development project in the previous survey. In most iterations, between 4% and 45% of developers who planned to use C++ did not.

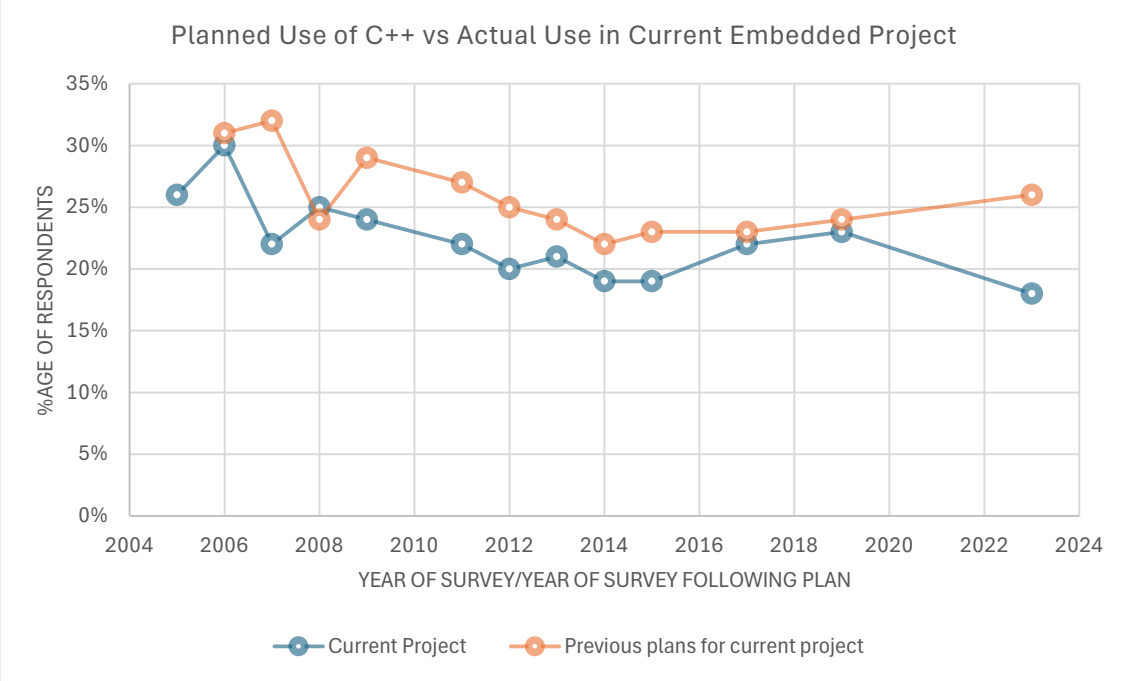


Figure 1.2: C++: Planned use vs Actual. The use of C++ as the language of the current embedded development project compared to plans – expressed in a previous survey – for its use in the next project. In general, actual use falls well below planned use, implying the presence of factors preventing its use. Data drawn from [190, 189, 9, 10, 37]. Responses were in answer to the questions 'My *current* embedded project is programmed mostly in:' and 'My *next* embedded project will likely be programmed mostly in:'.

Many reasons can been cited for this failure, including: perceptions of poor performance such as slow speed of compilation and execution, code bloat and lack of predictable performance and memory use [68, 66]. Additionally, developers are often not able to make a free choice between C and C++ because of constraints resulting from platforms and tool-chains, including availability and completeness of C++ compilers for the platform and C++ language support in project development tools and in libraries.

Since the introduction of the Arduino platform [159], there has been an increase in the use of C++ features for embedded platforms, including some extremely resource-constrained devices, using the Arduino programming environment and library ecosystem. However, the environment requires little C++ programming knowledge and Arduino source code uses a limited set of C++ features: the application environment can be viewed as a hybrid of C and C++.

"Coroutines were an essential part of early C++" [178], where they were used as

the basis for the task library in which "[t]he underlying facility is a simple and efficient tasking system with non-preemptive scheduling" implemented via a `task` class that maintains state and a system `scheduler` class [175]. However, coroutines were not introduced into the initial C++98 standard.

Gustafsson's proposal for resumable functions in 2012 [72] led to a lengthy debate [97, 99, 69, 98, 152, 131, 154, 155, 172] regarding the design of coroutines. As Stroustrup [178] wrote: "The design space for coroutines is huge, so consensus was hard to achieve." In particular, the questions of whether stackful or stackless designs – or both – should be supported and the use of dynamic memory allocation [171] were debated for many years. Finally, Nishanov's stackless design was accepted into the 2020 version of the standard [80], supported by its implementation's "superior performance" [178] in key use cases [85, 145], despite the fact that the design relied on dynamic memory allocation for a subset of use cases [133].

1.2 Summary of research

1.2.1 Motivation and research gaps

In the light of the inclusion of coroutines in the C++ 2020 standard, and of the fact that discussion of the coroutine implementation had been mostly focused on desktop, server and high-performance computing use cases, it was timely to consider the impact of coroutines on embedded system development. In particular, a research gap had appeared regarding the potential for coroutines to provide performant and ergonomic native solutions to the split-phase programming problem, as well as other asynchronous programming challenges that impacted embedded system development. This research was motivated towards reaching an understanding of the suitability and attraction of the proposed C++ coroutine implementation for embedded and Internet of Things developers.

The research would begin by identifying the research gaps (as detailed in the literature review contained within Chapter 2.1), and then continue by addressing the research questions below.

1.2.2 Research questions

The following research questions were identified initially and confirmed by the findings of the literature review:

RQ1: Can mainstream coroutine solutions apply to resource-constrained platforms?

RQ2: Are the costs deterministic?

RQ3: Can the benefits be clearly demonstrated?

These research questions would guide the investigation into the likely impact of the new coroutine implementation on the embedded and Internet of Things development area, paying attention to the issues that were found to be important to stakeholders.

Research question 1 was selected because, although there was found to be a demand for a language-native coroutine facility on resource-constrained devices, the public statements emerging from the C++ standards process had not included discussion or considerations specific to these platforms. The selection of research question 2 reflected the relative importance of deterministic memory and performance costs affecting embedded platforms and, in particular, resource-constrained devices and real-time systems. The third research question arose from the culture of the embedded development sector: if the benefits of C++ coroutines could not be very clearly demonstrated then there would be little likelihood of widespread uptake within the sector, given the substantial costs and perceived risks associated with such a change.

1.2.3 Research objectives

The objectives of this thesis are to:

- Provide the embedded system development sector with sufficient information to consider utilising C++ coroutines and, if necessary, to motivate migration from C to C++.
- Provide development tools, techniques or libraries to help industry write improved embedded software.
- Identify, within the C++20 standard and implementations, weaknesses or opportunities for improvement that specifically impact the embedded development sector.

1.3 Research chapters

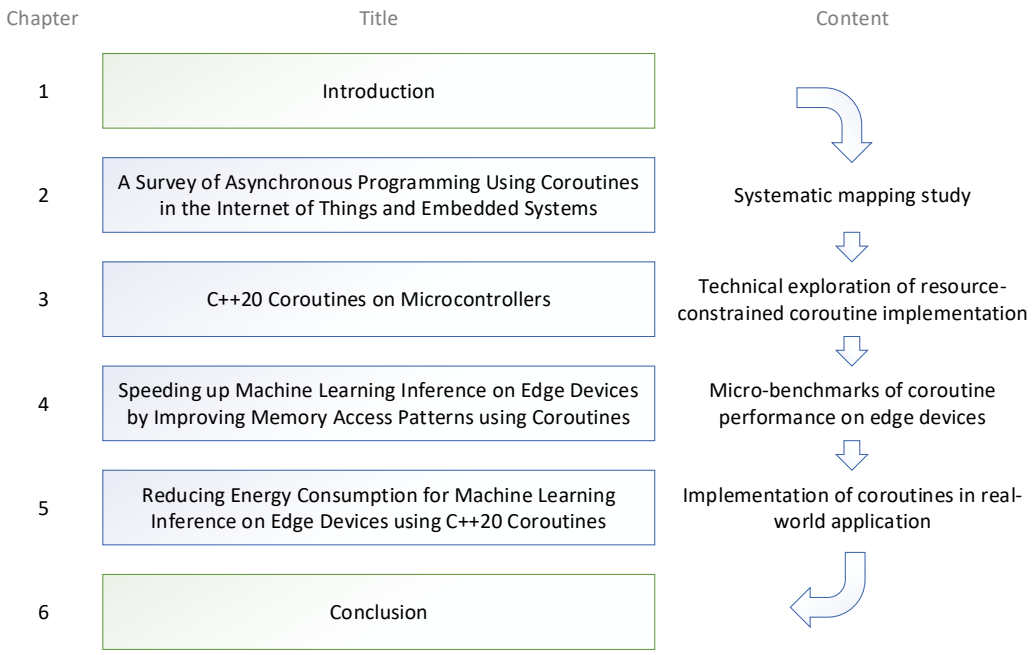


Figure 1.3: Chapter roadmap

Fig. 1.3 summarises how the four research chapters of this thesis move the viewpoint forward. The first research chapter is an investigation into the need for a mechanism such as coroutines for development of embedded systems on resource-constrained devices. This is followed by an analysis of the proposed C++20 coroutine implementation and its applicability to low-powered computing platforms. Next, the performance of C++ coroutines on various edge devices is studied using micro-benchmarks. The final research chapter is a case study that contains analysis and detailed measurement of coroutines within an edge device application.

1.3.1 Chapter 2 - A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems

Chapter 2 contains a traditional literature review and also a systematic mapping study [94], a form of systematic review that structures a research area as opposed to synthesising evidence. The study systematically analyses all the relevant academic literature, as summarised in Fig. 1.4. A search procedure was defined that queried the six main academic databases for the articles that referred both to the programming device (”coroutine” or ”light-weight thread”) and the platform (”internet of things”, ”embed-

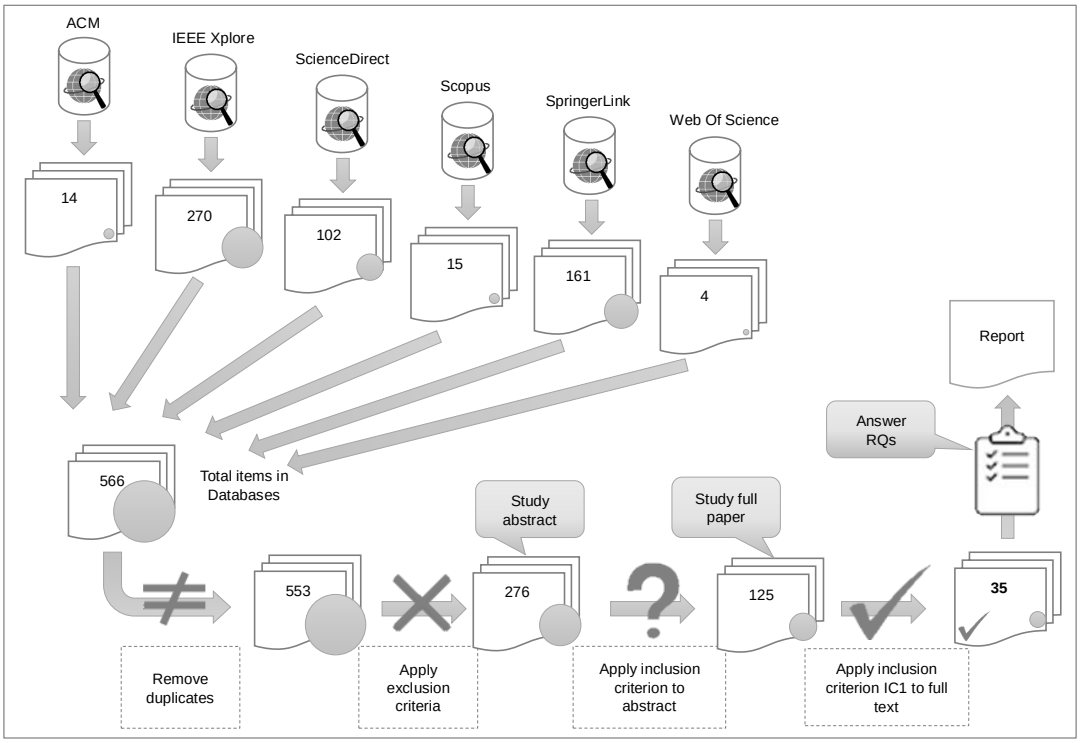


Figure 1.4: Chapter 2 - Summary of the search and selection process

ded systems", etc). After further inclusion and exclusion criteria were applied, the initial set of 566 papers was reduced to 35 papers, which were examined closely.

The study extracted key data from each paper, and built taxonomies of studies, addressing five core areas, as follows: (i) the software platform, including the programming language, the operating system and the method used to implement coroutines, (ii) the characteristics of the hardware platform, (iii) the use cases for coroutines and (iv) the intended benefits of coroutine use. (v) The details of the coroutine implementation were also collated, including important decisions such as whether control flow was managed on behalf of the programmer, whether the state of local variables was automatically managed, whether the coroutine was stackless or stackful [69, 152], and how the memory for the coroutine state was allocated.

These results were used to measure the demand for language-native C++ coroutines among developers on resource-constrained platforms, and to gauge which implementation features would be important to the developers. The study concluded from these outcomes that widespread demand existed for a language-native, well-supported C++ coroutine implementation on resource-constrained platforms, and that these devices would benefit from the implementation in a manner specific to their class.

1.3.2 Chapter 3 - C++20 Coroutines on Microcontrollers

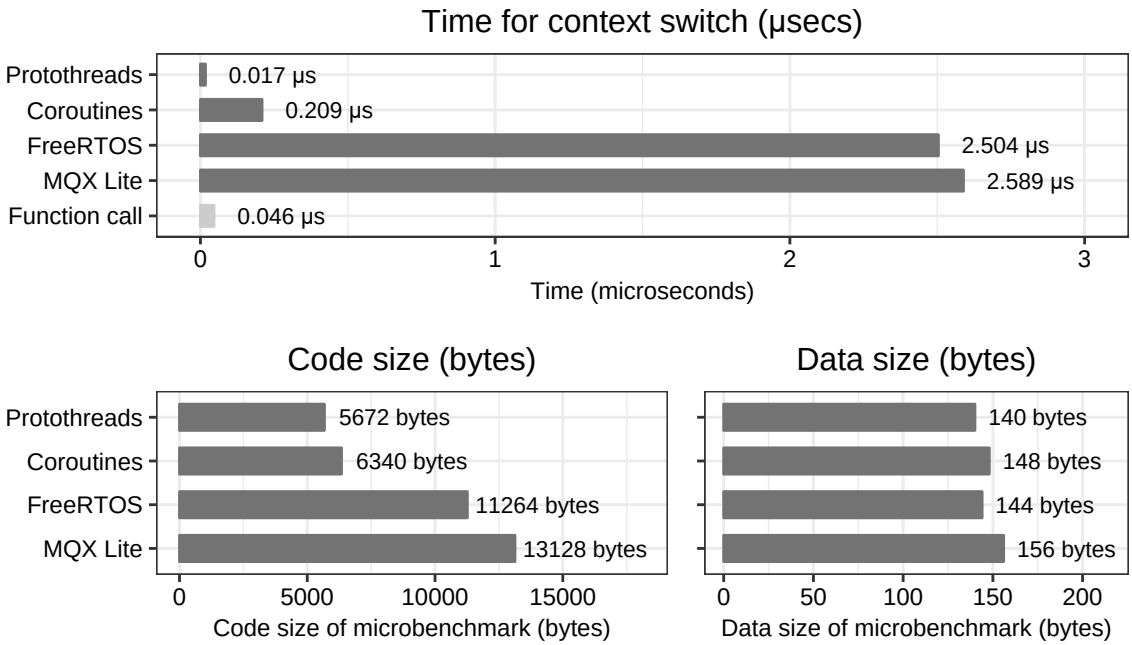


Figure 1.5: Chapter 3 - Summary of performance outcomes

The systematic mapping study of Chapter 2 found evidence of a demand for C++ coroutines on resource-constrained platforms. Chapter 3 moved forward to evaluate the C++20 coroutine specification [82] from the perspective of embedded systems developers.

The work offered three main contributions, as follows. First, we analysed the appropriateness of the C++ coroutines Technical Specification for embedded systems, an area which had not been discussed in the public debate. Second, as summarised in Fig. 1.5, we measured the performance of C++20 coroutines compared to commonly used alternative concurrency solutions: a hand-written state machine, the Protothreads library [49] and two leading real-time operating systems, FreeRTOS [15] and MQX Lite [135]. The measurements considered source lines of code, size of executable, run-time memory requirements and speed of performance. Finally, we investigated how easy it would be – from an application programmer's point of view – to use C++ coroutines on a constrained-resource platform (in this case, an ARM Cortex-M4 microcontroller development board). We analysed the complexity and the length of source code by building the firmware for a simple IoT device, both with and without coroutines.

A C++ implementation of the coroutine run-time library was iteratively designed, built and tested, satisfying the following criteria:

- Conform to the C++20 coroutine specification.
- Avoid using C++ exceptions, because they may significantly increase run-time memory usage and because they may introduce non-deterministic timing and memory behaviour [68].
- Do not use dynamic memory allocation, because the timing of such allocation is often non-deterministic [68] and because embedded systems are often resource-constrained with regard to memory.
- Avoid dependencies on the Standard Template Libraries (STL), because STL makes extensive use of both C++ exceptions and dynamic memory allocation.
- Fit within the very constrained code and data memory limits of the target platform.

The implementation of the library was a non-trivial task. While the STL contains classes that would provide useful infrastructure for coroutine calls, such as `std::future` and `std::promise`, these carry dependencies on C++ exceptions and dynamic memory allocation, and therefore had to be reengineered for a 'bare-metal' environment.

We concluded that it was not possible – within the constraints of the Technical Specification – to provide a software platform that would allow application programmers to deliver simple readable code that was certain to avoid dynamic memory allocation. Furthermore, when the memory was allocated globally or in the stack it proved impossible to determine memory requirements in advance, and a convoluted build cycle was required, since memory requirements only became fully determined at link time.

Our investigation into performance and development characteristics had more positive results. As can be seen in Fig. 1.5, the benchmark using coroutines ran 12 times faster than the benchmarks that used RTOS's, and the compiled executable code was half the size of the RTOS benchmark executables. The coroutine benchmark and the RTOS benchmarks used the same amount of data memory. The ergonomic benefits were measured by comparing the number of source lines of code (SLOC) contained in a sample application, excluding the generic source code of the coroutine implementation library. Compared to the equivalent standard FSM solution, this application code contained one sixth the number of SLOC.

We concluded that ”the proposed language enhancements potentially bring significant benefits to programming in C++ for embedded computers, but that the implementation imposes constraints that may prevent its widespread acceptance among the embedded development community”.

1.3.3 Chapter 4 - Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines

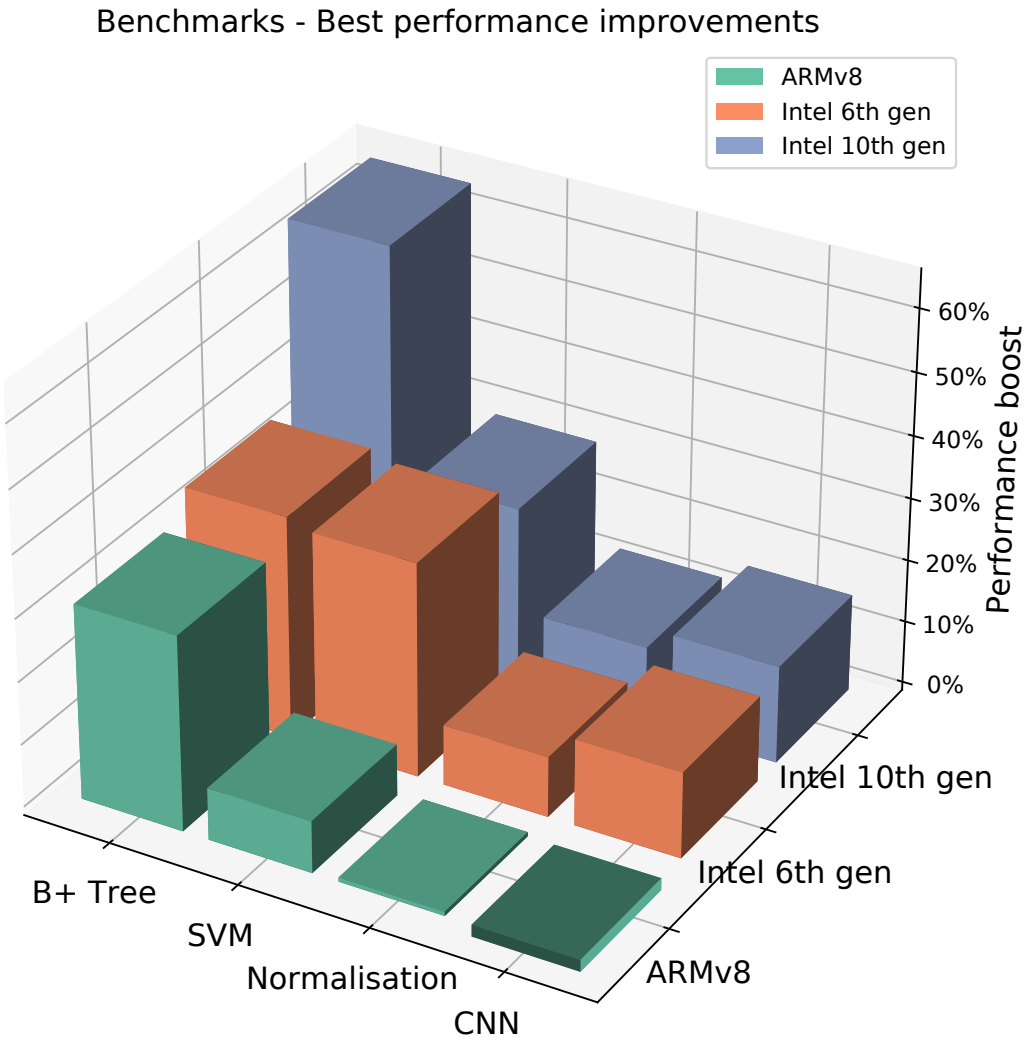


Figure 1.6: Best performance improvements achieved through coroutining and prefetching, across various algorithms and platforms

Chapter 3 concluded with a finding that the use of C++20 coroutines on embedded systems, while promising, remained constrained by its implementation with regard to memory allocation. Chapter 4 moved on from microcontrollers running on ‘bare metal’ with kilobytes of RAM to examine edge devices equipped with gigabytes of RAM and running Unix. The performance of coroutines on such edge devices was measured in

detail, using micro-benchmarks.

The benchmarks addressed tasks that are commonly deployed to edge devices, particularly within machine learning inference implementations, and included a local data search using a B+ tree, the inference stage of a support vector machine (SVM), the normalisation of a set of images to the ImageNet standard [43] and a convolutional neural network (CNN) inference using a 3x3 kernel.

The work compared the speed of performance of two execution patterns, applied to the iterative processing of a large set of data containing many smaller items. In the first case, the standard sequential model, the processing moves in a linear fashion from the start to the end of the data set. In the second case, the data is divided up into a number of subsets, and each subset is processed by a separate lightweight thread. Execution jumps from thread to thread, controlled by a round-robin scheduler. For these benchmarks, the lightweight thread mechanism was implemented using C++20 coroutines.

Additionally, a 'prefetch' technique may be used: a CPU cache load of the next 'chunk' of data is initiated just before control leaves the thread. By the time control returns to the thread, the prefetch has been completed and the data has been loaded into CPU cache: processing may continue without waiting for the data to be loaded. With or without the prefetch operation it is possible for code execution to benefit from this multi-threaded approach, because of improvements to the memory access pattern.

However, there exists a performance trade-off in this change to the execution pattern: against any benefit achieved through prefetching and improvements to memory access patterns, there is a cost to initiating a coroutine and in jumping between coroutines. The primary question addressed by these benchmarks was whether the coroutine infrastructure in the C++ language would be fast enough to provide a net benefit.

A second trade-off is that between any performance improvements achieved and the cost of the additional code complexity involved in splitting up a data set into sub-tasks and scheduling the sub-tasks into threads. The study also examined the complexity of the code required for this transformation.

The study explored a large parameter space: four benchmarked algorithms, three hardware platforms, two toolchains, four different numeric formats and a wide range of data set sizes, for a total of 23558 different tests. Each test was run with and without prefetching, and each was run 10 times, with outlers being removed.

As can be seen in Fig. 1.6, impressive speed improvements were recorded: up to 65% for the B+ tree benchmark and as much as 34% for the SVM. Improvements in the CNN and normalisation benchmarks were less marked but still substantial at 15.5% and 12% respectively.

1.3.4 Chapter 5 - Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines

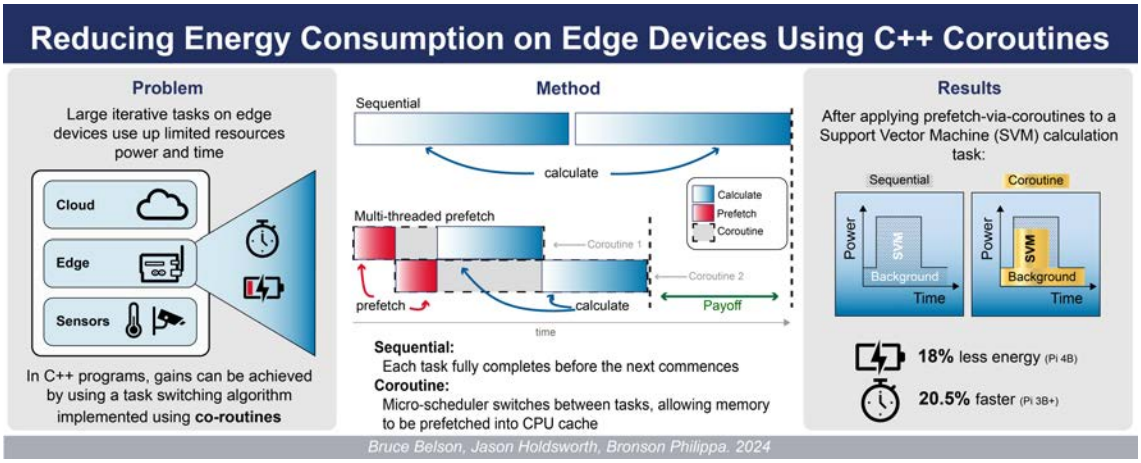


Figure 1.7: Graphical summary for Chapter 5

Following the substantial speed improvements documented in Chapter 4, the thesis concluded with a study of the use of coroutines in a real-world C++ application in Chapter 5.

The case study consisted of a Prognostic Health Maintenance (PHM) running on a Raspberry Pi which receives streams of vibration data as envelope spectra from the nodes of a wireless sensor network (WSN) and processes them locally through a collection of Support Vector Machines (SVMs).

We applied a simple transformation to the application code, using C++20 coroutines to reorganise the execution order of the SVMs. We continued to see the speed improvements that were predicted by the micro-benchmarks of Chapter 4, with execution time for the SVM reduced by as much as 20.5%. Furthermore, we also recorded reduced energy consumption: the energy used by the SVM task could be reduced by up to 18% and the peak power level was reduced by up to 4%. (The transformation and its outcomes are summarised in Fig. 1.7.)

The speed improvements can translate into a reduction of the number of edge de-

vices required by the network, with consequent savings in the cost of equipment and deployment. The net savings in energy could translate into longer battery life, offering savings in deployment and maintenance, or – for a mobile platform – a reduction of device weight.

1.4 Notes

The four research chapters in this thesis were prepared for three different publishers, using four different styles. The presentation in this thesis has been standardised: all chapters use a single-column format, and all text fonts are shared between chapters. However, no attempt has been made to standardise the colours, styles or themes used for the graphical components, and these remain in their original form.

Citations and references are now held in common between chapters, share a common format (IEEE), and are presented in a single bibliography at the end of the entire document. The bibliography is ordered by first author surname, with a limit of three author names.

The source code for all the chapters has been made available on-line in public repositories, along with experimental results. The URL for each repository can be found in the respective chapter.

The thesis (and all four original papers) were prepared using *TeXstudio*, *pdflatex* and *biblatex*.

Chapter 2

A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems

This chapter is a reformatted version of the following paper in the ACM journal *ACM Transactions on Embedded Computing Systems (TECS)*.

> B. Belson, J. Holdsworth, W. Xiang and B. Philippa, "A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems" in *ACM Transactions on Embedded Computing Systems*, Volume 18, Issue 3, May 2019, Article No.: 21, pp 1–21, doi: 10.1145/3319618.

The paper was published in May 2019, before the inclusion and specification of coroutines in C++20.

Chapter Abstract

Many Internet of Things and embedded projects are event-driven, and therefore require asynchronous and concurrent programming. Current proposals for C++20 suggest that coroutines will have native language support. It is timely to survey the current use of coroutines in embedded systems development. This chapter investigates existing research which uses or describes coroutines on resource-constrained platforms. The existing research is analysed with regard to: software platform, hardware platform and capacity; use cases and intended benefits; and the application programming interface design used for coroutines. A systematic mapping study was performed, to select studies published between 2007 and 2018 which contained original research into the application of coroutines on resource-constrained platforms. An initial set of 566 candidate papers, collated from on-line databases, were reduced to only 35 after filters were applied, revealing the following taxonomy. The C & C++ programming languages were used by 22 studies out of 35. As regards hardware, 16 studies used 8- or 16-bit processors while 13 used 32-bit processors. The four most common use cases were concurrency (17 papers), network communication (15), sensor readings (9) and data flow (7). The leading intended benefits were code style and simplicity (12 papers), scheduling (9) and efficiency (8). A wide variety of techniques have been used to implement coroutines, including native macros, additional tool chain steps, new language features and non-portable assembly language. We conclude that there is widespread demand for coroutines on resource-constrained devices. Our findings suggest that there is significant demand for a formalised, stable, well-supported implementation of coroutines in C++, designed with consideration of the special needs of resource-constrained devices, and further that such an implementation would bring benefits specific to such devices.

2.1 Introduction

The Internet of Things (IoT) [2, 11, 70] continues to grow both in the scale and variety of attached devices and in the number of developed applications [117, 122]. This growth draws attention to the software engineering of the resource-constrained embedded systems that are a frequent component of heterogeneous IoT applications. Such attention is

all the more urgently required because of new challenges with regard to security [166], reliability [70] and privacy [193].

Many IoT and embedded systems have an event-driven architecture; their software is consequently implemented in an asynchronous programming style, whereby multiple tasks wait on external events. Asynchronous code is challenging to write because application logic becomes split between the function initiating the request and the event handler that is invoked when the response is ready [62, 107, 121]. This "split-phase" architecture becomes increasingly complex when the developer introduces more event sources (such as timeouts) with their own event handlers. There may be interaction between various split-phase events, which can add degrees of freedom to the various state models: consequently there is an increasing likelihood that the source code addressing a single event is split between separate locations, forcing the reader to jump between them. Application logic is obscured by the split-phase fragmentation, leading to a gap between the design of the system and its source code representation, making the code harder to understand and more difficult to maintain [24, 52, 116, 88].

A solution to the split-phase problem for desktop software has been language support for coroutines [36, 96, 119] and promises [24, 111, 116]. For example, in C#, JavaScript, and Python, developers can use an "await" keyword to wait on an external event. This means that asynchronous code can be written just as clearly as the equivalent code in a synchronous style that uses blocking code. However, resource-constrained embedded systems are overwhelmingly programmed in C or C++ [9, 168], which lack support for the "await" pattern.

The C++ standardisation committee is currently debating the inclusion of coroutines, and at least two competing designs have been proposed [81, 154]. The addition of coroutines to C++ would create an opportunity to simplify embedded systems code. Existing research on coroutines in C++ may not have considered the needs of embedded systems and other extremely resource-constrained devices, because the initial implementations used compilers that do not target such platforms [124]. Here, we specifically focus on small embedded systems that have insufficient memory to run Linux or another general purpose OS. If the C++ language adds the async/await and coroutine patterns, we believe it is important that the needs of resource-constrained platforms are also considered.

This chapter contains a systematic mapping of the use of coroutines in embedded systems, conducted by searching academic databases and manually inspecting every matching paper. It thus provides a complete perspective on academic research addressing the use of coroutines in embedded systems to inform the C++ standardization process by identifying how and why coroutines are used. The study uses the mapping to build a taxonomy of existing research with regard to platform, use cases and implementation.

The design of the study, details of the methodology used for each stage and the results of each stage are available in spreadsheet format. The remainder of this chapter is organised as follows. Section 2.2 contains the background, beginning with an introduction to the development environment for C/C++ programs on resource-constrained devices, to some of the problems commonly encountered by developers and to the types of solution currently applied to these problems. It continues with a discussion of the use of coroutines in C and C++. Section 2.3 details the methodology of the mapping process used in this study, some of the logic underpinning the methodological choices, and a review of related work. Section 2.4 explores the results and presents insights. Section 2.5 contains a discussion of results and an analysis of research gaps. Section 2.6 discusses further research possibilities and concludes the chapter.

2.2 Background

2.2.1 Async/Await pattern

Much of the program flow in IoT and embedded device programs is asynchronous, for example, requiring the software to wait on responses from a remote device or sensor. A naïve approach to implementing this flow results in complex arrangements, such as a finite state machine (FSM) and multiple fragments of code. This produces source code that is complex, fragile and inflexible.

Alternatively, there are two common patterns for a simpler and more robust design. The first, continuation-passing style (CPS), which is seen commonly in JavaScript, can lead to the ”pyramid of doom” or ”callback hell” phenomenon [24, 52, 116, 88] when multiple sequential operations are composed.

A more elegant approach is the async/await pattern [20, 136, 182], which is a pop-

ular device for transforming continuation-passing style code into direct programming style, with all the asynchronous steps of a sequence written in a single ordered sequence within a single block of code. The pattern has been used successfully in several languages, notably C# [20, 136] and JavaScript, as part of the ECMAScript 2018 proposal [51]; in C++, proposals are currently being considered for inclusion in the C++ 2020 standard, using new keywords 'co_await', 'co_yield' and 'co_return' or alternative syntax [81, 154]. The async/await pattern allows the programmer to write a single continuous set of statements in a direct programming style, which will be performed in the correct order, even when they are run asynchronously as a set of separate events. Furthermore, the pattern avoids the explicit use of global variables.

2.2.2 Coroutines

Coroutines extend the concept of a function by adding suspend and resume operations [36, 96, 119]. Coroutines can be used for a variety of purposes including (i) event handlers [49]; (ii) data-flow [100]; (iii) cooperative multitasking [180] as well as (iv) the async/await pattern [81].

During suspension, the implementation stores the execution point of the coroutine, and usually (but not always) the state of local variables. For example, Protothreads [49] is a coroutine implementation for embedded systems where local variables are not preserved: instead all variables within the coroutine must be statically allocated. This strategy reduces the overhead of context switching and provides predictable memory usage but produces coroutines that are non-reentrant. Furthermore, code defects are more likely when the programmer is responsible for explicitly managing coroutine state. This study will examine both types of coroutine in the context of embedded systems.

Coroutine implementations may be further categorised into stackful or stackless types. A stackful coroutine has its own stack which is not shared with the caller, and hence local variables can be stored there during suspension. Conversely, a stackless coroutine pops its state off the stack during suspension (like a normal function return). For stackless coroutines, other mechanisms must be introduced in order to preserve state, such as storing local variables in global storage.

Furthermore, a stackless coroutine often can only be suspended from within the coroutine itself and not from a subroutine (i.e. a function called from the coroutine).

For example, C++ proposal N4680 is a stackless model that requires all yield or return statements to be contained within the body of the coroutine.

Neither model is considered universally appropriate for the various C++ use cases [69, 152]. Alternative techniques, such as stack slicing, have been used to preserve state in a stackless implementation and provide single threaded cooperative multitasking [187, 186].

2.2.3 Previous coroutine implementations for constrained platforms

Early implementations used macros in C to add coroutine-like features. For example, Duff's device takes advantage of the fall-through behaviour of C's case statement in the absence of a break statement [46]. It is unusual in that a block such as do ... while, can be interleaved within the case statements of a switch statement. Tatham [184] described a coroutine solution in C, which makes use of Duff's device to efficiently implement coroutines through macros, without the need to explicitly code a state machine. However, Tatham noted that "this trick violates every coding standard in the book" and Duff called the method a "revolting way to use switches to implement interrupt driven state machines". This technique was extended by Dunkels et al. for Protothreads [49], which provided conditional blocking operations on memory-constrained systems, without the need for multiple stacks, and formed the core of the widely used real-time operating system Contiki [47].

Protothreads (and any other solution based on Duff's device) can be considered to suffer from two serious defects. First, their use adds a serious constraint to C programs: switch statements cannot be used safely in programs that use Protothreads; they may cause errors that are not detected by the compiler but cause unpredictable behaviour at run-time. Second, they do not manage local variable state on behalf of the programmer: any variable within the coroutine whose state should be maintained between calls must be declared as static (global) [48]. This has consequences for reentrancy, and for code quality. On the other hand, they are an extremely cheap solution in terms of coding effort, memory use and speed, and they are portable, because they use pure C. The original library is written in C; it has been ported to C++ [138].

Listing 2.1 contains a fragment of code that used Protothreads to implement part of an asynchronous producer/consumer pattern. Listing 2.2 shows a similar code frag-

Listing 2.1: Fragment of Protothreads code for asynchronous producer/consumer threads

```
static struct pt_sem full, empty;

static
PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;
  PT_BEGIN(pt);
  for (consumed = 0; consumed < NUM_ITEMS;
  ++consumed) {
    PT_SEM_WAIT(pt, &empty);
    consume_item(get_from_buffer());
    PT_SEM_SIGNAL(pt, &full);
  }
  PT_END(pt);
}
```

Listing 2.2: C++ code fragment using co_await for asynchronous producer/consumer threads

```
task<> consumer(semaphore& sem) {
  auto producer = async_producer(sem, NUM_ITEMS);
  for co_await(auto consumed : producer) {
    consume_item(get_from_buffer());
  }
}
```

ment, this time using C++ language features, including the co_await keyword of the current C++ standardisation proposal N4680. We observe several differences between the two. Listing 2.2 contains fewer lines of code than Listing 2.1; Listing 2.2 does not contain macros; Listing 2.1 requires that local variables be declared as 'static', but Listing 2.2 does not. While both code fragments present conceptual changes from the synchronous equivalent, we believe that the change in Listing 2.2 is more transparent and more clearly presented.

In 1992, Gupta et al. examined a coroutine-based concurrency model for resource-constrained platforms as part of a comparison between alternative models [71]. In 2000, Engelschall summarised the various techniques based on setjmp & longjmp [57]. FreeRTOS [15] is an open source real-time operating system developed "around 2003" that contains a coroutine scheduler: local variable state is not maintained. In 2006, Rossetto and Rodriguez described a new concurrency model [156] implemented as an extension to TinyOS [108], using coroutines as the basis of the integration; the implementation is stackful and local variables' states are maintained.

Schimpf (2012) [162] provides a modified version of Protothreads which supports a priority-based scheduler. Cohen et al. (2007) [34] provide a coroutine-based scheduler for TinyOS [108] which is used to implement "RPC-like interfaces"; these support a direct programming style for communications code written in nesC [62]. Riedel et al. (2010) [151] generate C code for multiple platforms, including a version that uses coroutines to provide concurrency. Susilo et al. (2009) [180] use a coroutine-based scheduler to achieve "[r]eal time multitasking [...] without interrupts". Finally, Andersen et al. (2017) [6] reject the use of C++ futures because the implementation model needs to handle a stream of events, rather than a single event, and is therefore non-deterministic in its use of memory, which is undesirable on a constrained platform: "One is forced therefore to trade off the reliability of promises [...] in order for them to work in the embedded space." Instead, the authors use callbacks for C++ event handling.

The scripting language Lua possesses a coroutine implementation [41] and has been successfully used on microcontrollers [76]. MicroPython [63] is a Python 3 version which supports microcontrollers [64] and includes support for generators and coroutines [157, 185].

2.2.4 Programming Languages: C and C++

The majority (78%) of embedded systems are programmed in C or C++ according to the 2017 Embedded Markets Study [9]. The C language is the most popular, but its usage is slowly declining over time in favour of C++ and other languages. Between 2015 and 2017 the proportion of embedded projects using C++ rose from 19% to 22%, while C use fell from 66% to 56%. Coroutines are proposed for the C++ language, not C, so embedded programmers would need to use C++ to access these features. We believe that C++ usage will continue to increase, and therefore the design of C++ coroutines should consider the constraints of embedded software.

The switch from C to C++ need not be dramatic. C++ is close to being a superset of C [176]. With the right compiler support, it is possible to migrate an embedded codebase from C to C++ merely by changing a compiler flag. There are potential problems with the migration from C to C++, including the possibilities that the code produced may be larger, slower and less likely to contain blocks that are appropriate for placement in ROM than the C code [68, 77]. A more subtle problem is that the code may

be less amenable to worst-case execution time (WCET) analyses. Goldthwaite [68] examined these problems, and identified three areas where difficulties might exist despite defensive programming, all of them with regard to timing analysis: (i) dynamic casts, (ii) dynamic memory allocation and (iii) exceptions.

A number of further problems related to tool-chains and platforms may inhibit migration to C++. Many hardware platforms are closely related to specific tool-chain implementations. Sometimes the only compiler available is a manufacturer's proprietary version, for which C++ support may be unavailable (such as MPLAB XC8 for the PIC 8-bit devices[1]), or may not include support for recent C++ standards and features (such as the commonly used Keil ARM compiler[2], which does not support C++20 as of January 2025). The libraries that provide a specific hardware access layer may not include support for C++ 'out of the box'. The project management systems that generate and manage application files may not support C++. When the Standard Template Library cannot be used because of its dependencies on exceptions and dynamic memory allocation, there may not be an appropriate library to use as a substitute.

The features that might persuade a development team to make the move to C++ have always included well-known front-end features such as namespaces, encapsulation, and inline functions, all of which offer benefits regarding code clarity but have no implementation cost in terms of code size or speed. Replacing split-phase functions with a direct programming style by using new, widely supported, language standards would appear to be a strong enticement for developers to migrate. It remains to be seen whether the feature can be provided for embedded systems without including two of the three behaviours which Goldthwaite [68] identified as being problematical: dynamic memory allocation and exceptions.

[1] https://ww1.microchip.com/downloads/aemDocuments/documents/DEV/ProductDocuments/UserGuides/MPLAB-XC8-C-Compiler-Users-Guide-for-PIC-DS50002737.pdf

[2] https://developer.arm.com/documentation/101458/2404/Standards-support/Supported-C-C---standards-in-Arm-C-C---Compiler

2.3 Systematic mapping study

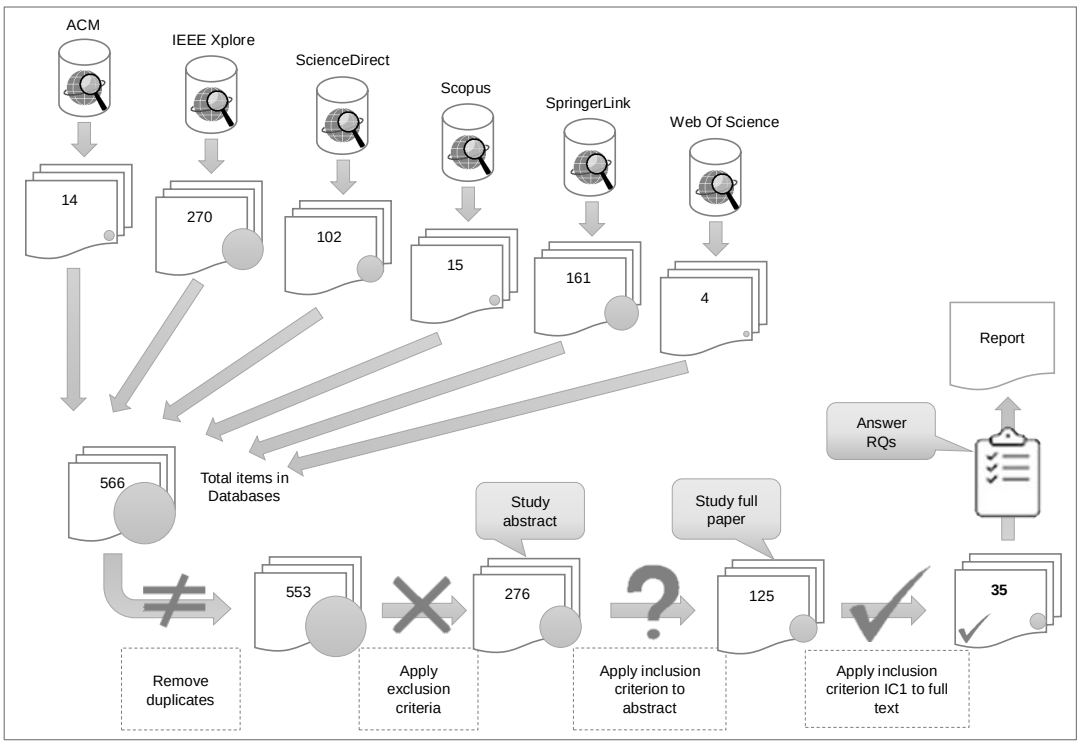


Figure 2.1: Summary of the search and selection process

2.3.1 Overview

This systematic mapping study is informed by the guidelines of Kitchenham [93], Kitchenham and Charters [94] and Petersen et al. [143]. The process is illustrated in Figure 2.1. The process searched six online databases, selected for relevance [23] and availability, for papers containing a term from each of the lists in Table 2.1. We ensured completeness by iterative testing using snowballing [95, 142] and by careful handling of database-specific behaviours regarding plurals, spellings and abbreviations.

Table 2.1: Search strings used for online databases

Part 1: Pattern		Part 2: Platform
coroutine OR ”lightweight thread”	AND	IoT OR ”Internet of Things” OR ”Cyber Physical Systems” OR RTOS OR ”Real-time Operating Systems” OR ”Embedded Systems” OR WSN OR ”Wireless sensor networks” OR WSAN

2.3.2 Search procedure

The search procedure first applies each inclusion criterion (IC), referred to as IC*s* where *s* is an alphanumeric suffix. The data base search facilities are used to collate all papers which satisfy all ICs. Next each exclusion criterion (EC), referred to as EC*n* where *n* is numeric, is applied: any item which fails any EC is removed from the set. The full list of criteria can be found in Tables 2.4 and 2.5 in the appendix.

The main inclusion criterion (IC1) was that the paper should contain original research into the application of coroutines on resource-constrained platforms. This criterion excluded a large body of papers which applied coroutines only within the simulation of resource-constrained platforms, not on the platform itself. For clarity, the exclusion of papers using coroutines only in the sumulation was stated explicitly in a secondary inclusion criterion (IC1a).

The exclusion criteria were informed by previous studies [94, 143]. Papers were excluded if they lacked a scholarly identifier such as DOI or ISBN (EC1) or an abstract (EC2), were published before 2007 (EC3), were not written in English (EC4), were not available to the reviewers (EC5), were earlier versions of another paper (EC6), were not primary studies (EC7) or were not in any of the selected publication classes (journal articles, conference papers or book chapters) (EC8).

Two searches were conducted in October 2017 and in September 2018 across all databases. These searches resulted in 187 journal articles, 224 conference papers and 155 book chapters. This informed our decision to include all three publication classes within the search domain. The decision was made to include only studies published since 2007; this criterion excluded approximately 43% of the original search results.

Details of the search strings, inclusion and exclusion criteria, procedures and download scripts can be found in the supplementary materials.

2.3.3 Other systematic reviews and mapping studies

An initial tertiary study was executed, being a review of existing reviews and mapping studies in the area of interest, as suggested by Kitchenham and Charters' guidelines [94]. The work concluded that, at the time of writing, this study appears to be the first to systematically map the use of coroutines in resource-constrained systems, whether

embedded systems or IoT component systems.

2.3.4 Research questions

A major motivation for the study was to prepare the ground for an acceptable implementation of the await/async and generator patterns on resource-constrained platforms, using coroutines. The research questions therefore address what is known about hardware and software platforms, developer preferences, use cases, intended benefits, and application programming interfaces (APIs).

Research question 1 (RQ1) investigated the software platform, including the programming language, the operating system and the implementation used for the relevant language feature. Research question 2 (RQ2) looked at the hardware platform, including memory size and processor family. Research questions 3 and 4 (RQ3 and RQ4) assessed the use cases and intended benefits respectively of the coroutine usage. Research question 5 (RQ5) assessed the programming interface.

The research questions are listed in full in Table 2.2. By examining the hardware and software characteristics of previous implementations we aimed to identify the salient characteristics of the environment within which a coroutine implementation must function. By investigating use cases and desired outcomes, we would identify some of the necessary characteristics of a successful implementation. Finally, by examining the programming interface we hoped to observe how researchers addressed some of the design trade-offs of the implementation.

2.3.5 Threats to validity

Data extraction followed the principles laid down in Petersen et al. [142] for repeatability.

The validity of the results of this study are exposed to multiple sources of threat, particularly with regard to (i) study selection, (ii) data extraction and (iii) classification.

During study selection, the search process was recorded in detail and the search strings were tested for repeatability and for consistency across databases. Snowballing describes the process of expanding the search results by recursively selecting papers that are cited by a selected paper or those that cite a selected paper [95, 142]. While the study did not utilise snowballing during the final search process, it did use it during

Table 2.2: Research questions

Code		Research question
RQ1		What was the software platform?
	RQ1a	What was the programming language used?
	RQ1b	What method was used to implement coroutines?
	RQ1c	What was the operating system used (if any)?
RQ2		What was the hardware platform?
	RQ2a	What was the class of hardware platform?
	RQ2b	How much read-only or flash memory (ROM) was available?
	RQ2c	How much random-access memory (RAM) was available?
	RQ2d	What was the processor family?
	RQ2e	Was the processor 8-bit, 16-bit or 32-bit?
	RQ2f	What was the processor's instruction set?
RQ3		What were the use cases?
RQ4		What were the intended benefits of using coroutines in this context?
RQ5		What is the API of the coroutine?
	RQ5a	Does the paper discuss an implementation of coroutines?
	RQ5b	Is the control flow managed on behalf of the developer?
	RQ5c	Is the state of local variables automatically managed?
	RQ5d	Is the coroutine implementation stackless or stackful?
	RQ5e	How is the coroutine state allocated?

the earlier stages of establishing search strings, and some searches were consequently amended. During the application of selection criteria, the reviewers (B. Philippa and B. Belson) conferred whenever differences arose, and periodically discussed and reviewed the processes being used, using both contentious cases and randomly selected test papers to compare individual processes.

The guidelines of Petersen et al. [143, 142] were followed with regard to the data extraction process: a data collection form was constructed in Excel, and was used consistently to record the process, in order to improve repeatability and accuracy, and to reduce subjectivity.

To improve the consistency of classification a subset of papers was inspected by both reviewers, and the classifications were compared and discussed. This comparison was iterated until the rationale for classifications was fully established and any contentious cases had been resolved.

2.3.6 Data set

The initial search found 566 results; removal of duplicates left 553 documents. More than half of these failed the exclusion criteria, leaving 276 whose abstracts were studied.

Approximately 55% of the surviving studies immediately failed the inclusion criteria, leaving 125 to be studied in full. After applying the inclusion criteria on the basis of the entire text, about 72% failed, and 35 studies were retained [4, 5, 7, 6, 19, 21, 33, 34, 50, 55, 58, 60, 67, 79, 83, 84, 87, 89, 91, 100, 101, 113, 115, 126, 130, 134, 137, 140, 151, 162, 173, 179, 180, 73, 196]. Of these, 21 studies included a discussion of the implementation of coroutines. The lower half of Figure 2.1 illustrates the process.

The selected papers addressed the issue of coroutines despite the lack of mainstream language support. These researchers identified a need that was not addressed by common languages and showed the potential benefits of these features. Now that native asynchronous programming support is being added to the C++ language, it is likely that demand from embedded software developers will only increase.

Table 2.7 in the appendix section provides the complete list of selected studies.

2.4 Results

2.4.1 Overview

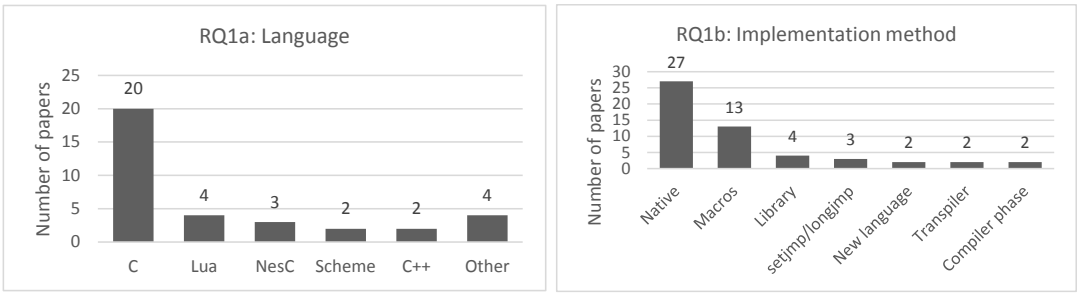
The research identified 35 papers of relevance, of which 21 described coroutine implementations, developed in 7 different programming languages. The results are briefly presented below. The detailed lists of results can be found in the supplementary materials.

2.4.2 Programming language

C was the predominant programming language, as shown in Figure 2.2a. A total of 20 papers (57%) used C, and a further 5 papers (14%) used related languages (C++ and NesC). Lua was the next most common language used, with 4 papers.

2.4.3 Coroutine implementation

To implement coroutines, 27 papers (77%) used a native method, i.e. avoiding techniques that required a new or changed tool chain. In native implementations, 13 papers employed macros (of which 7 were based on Duff's device) and 4 used libraries; in 3 papers ([196, 34, 87]) the C setjmp/longjmp language device was used.



(a) RQ1a - Programming languages used (b) RQ1b - Coroutine implementation

Figure 2.2: Language outcomes

Several studies extended the tool chain or created a new tool. Two papers contributed new languages [84, 58], and one paper [130] provided a set of language extensions. Two papers employed a transpiler that translates from one language to another - one from Lustre to OCaml [84] and one from a synchronous extension of C to standard C [89]. One paper [60] used a precompiler, and one paper [83] provided a new compiler optimisation phase.

Two studies called out to another language to implement the coroutines: one [140], written in the Lua language [41], directly manipulated the hosting environment through the C API; another [91] used non-portable assembly language. The results are summarised in Figure 2.2b.

2.4.4 Operating system

Of the 26 application instances studied (taken from 25 papers) that were written in C, C++ or NesC, 13 (50%) used (or extended) a widely-known embedded operating system (Contiki [47], TinyOS [108] or FreeRTOS [15]) and 9 (35%) used a unique operating system, or one that was generated for each application, as shown in Figure 2.3. There was not enough information in the papers themselves to judge how many of these 9 papers could be considered 'bare-metal'.

2.4.5 Memory

Figure 2.4a shows the ROM and RAM sizes of the selected platforms, using logarithmic scales. As observed in RQ2a, there were many systems with low RAM sizes: the median value was 10 kb. There was a positive correlation (r=0.64) between ROM size and RAM size.

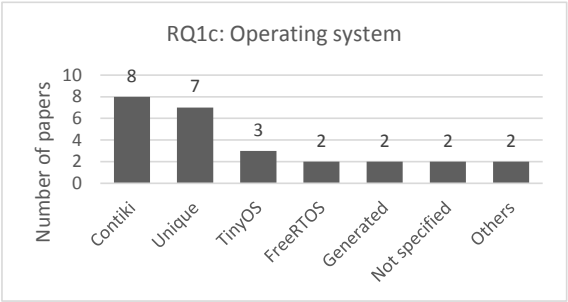
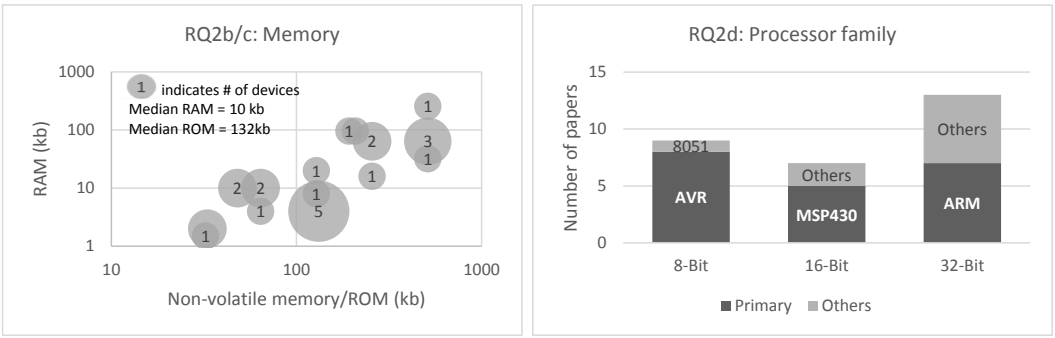


Figure 2.3: RQ1c - Operating systems used in selected studies using C-like languages



(a) RQ2b/c - Memory

(b) RQ2d - Processor families by bit width

Figure 2.4: Hardware outcomes

2.4.6 Processors

Only 45% (13 out of 29) of the CPUs that were identified were 32-bit processors: 9 were 8-bit and 7 were 16-bit. The fact that 55% (16 out of 29 studies) used 8- and 16-bit devices indicates that coroutines are applicable to very constrained platforms.

It is also notable that within the 8-bit segment, all but one were of the megaAVR family; among 16-bit processors 5 out of 7 used the TI MSP430 architecture. Within the 32-bit segment the picture was less clear-cut: just over half used the ARM architecture. These types of microcontrollers are widespread in IoT and embedded systems [9]. These results are summarised in Figure 2.4b. Full details are in the supplementary materials.

2.4.7 Use cases

The four most common use cases were concurrency (49% of papers), network communication (43%), sensor readings (26%) and data flow (20%), as illustrated in Figure 2.5a. It is notable that all four of these use cases are often considered to present difficulty or complexity for programmers. (See the supplementary materials for details of use cases and their classifications.)

(a) RQ3 - Use cases (b) RQ4 - Intended benefits

Figure 2.5: Usage outcomes

These use cases are common across many platforms, and not just resource-constrained devices. Syntax designed for desktop systems is likely to handle these cases relatively well. Contrasting these use cases with those found in desktop development, we observe that user interfaces (a strong driver of coroutines in desktop and portable system development) are absent and that sensor readings (a rare requirement in desktop systems) are prominent.

2.4.8 Intended benefits

Of the intended benefits the most common classifications were (i) code style and simplicity (34%), (ii) scheduling (26%) and (iii) efficiency (23%), as summarized in Figure 2.5b. (The supplementary materials contain details of the classifications of benefits.)

We have observed that split-phase programming leads to error-prone, hard-to-maintain code; it is therefore unsurprising that code style and simplicity leads the list.

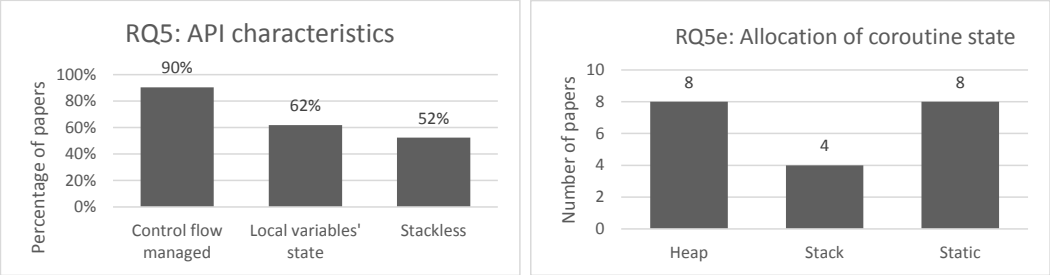
However, the popularity of scheduling as a benefit of a coroutine implementation is not mirrored in mainstream desktop programming, and it may therefore not figure high in the priorities of the C++ language specification process. Coroutines provide a tool with which to build schedulers, and many embedded software applications must provide their own scheduler, either because of the special requirements of the device [79, 140, 180] or to minimize code size by providing only the minimum requirements.

The high incidence of efficiency as an intended benefit also reflects the latency constraints of embedded systems.

2.4.9 Application programming interface

Of the 35 studies analysed, 21 discussed an implementation of coroutines: the questionnaire results for RQ5 are listed in the supplementary materials.

The API questions (RQ5b-e) could not in all cases be answered directly from inspection of the papers. In these cases, unless the answer could be found in the supplementary materials, linked source code, or was well-known to the researchers, the question was answered 'Unknown'. In some cases the source code referenced by the paper was no longer available.



(a) RQ5 - API characteristics (b) RQ5e - Allocation of coroutine state

Figure 2.6: API outcomes

Figure 2.6a summarises the basic API characteristics for those studies which examined an implementation of coroutines. The overwhelming majority (89%) of implementations managed control flow on behalf of the programmer; more than two-thirds managed the state of local variables. The outcome with regard to stackless and stackful implementations was more balanced: 11 stackless versus 8 stackful.

We have observed that managed, deterministic use of memory is a common requirement for embedded systems: in Figure 2.6b we see that over a third of papers (8 of 21) supported the allocation of coroutine state on the heap, which is not appropriate for embedded systems, and 4 used the stack, which may not be appropriate if the state size is large or of a size unknown at compile time.

2.5 Analysis and discussion

2.5.1 Analysis of API design

Figure 2.7 examines the API characteristics of the various implementations, grouped into (i) native C/C++, (ii) non-native C and (iii) languages other than C, where non-native C

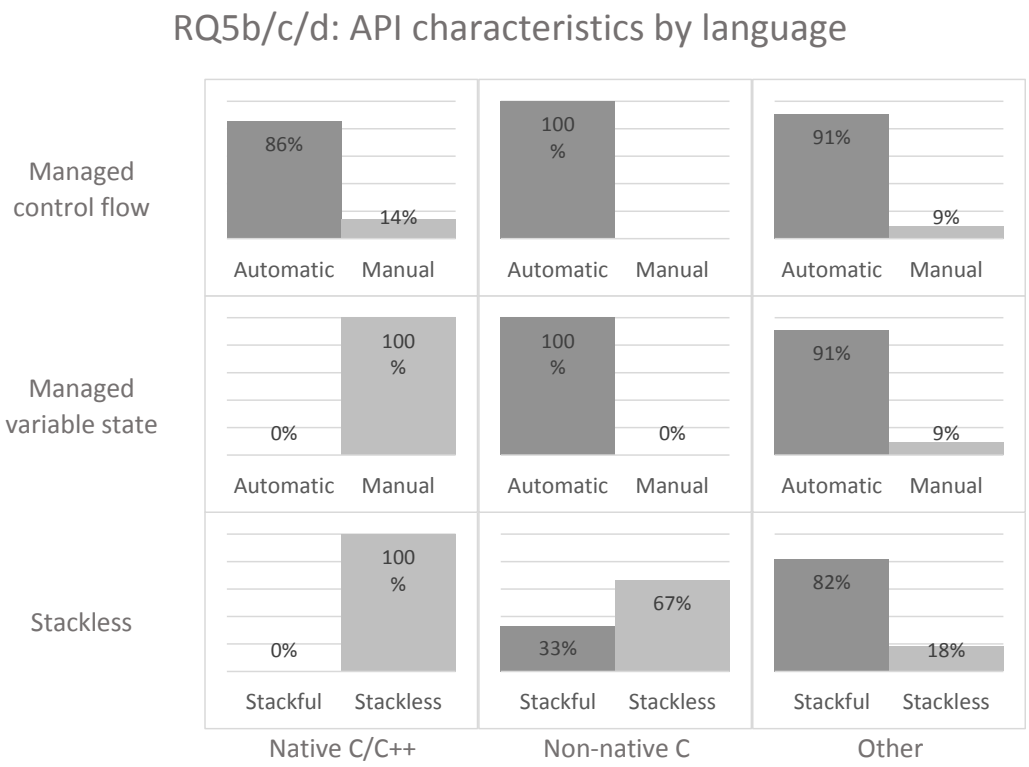


Figure 2.7: RQ5b/c/d - API characteristics by language

refers to language extensions, transpilers, or tools that otherwise extend the C compiler tool chain. The results for languages other than C and for non-native C implementations are interesting because they may reveal what the language designers and implementers considered to be desirable characteristics. (In each case the percentage shown is a fraction of the unique implementations inspected; it is not necessarily representative of the population at large.)

This chapter has suggested that the management of control flow on behalf of the programmer (RQ5b) is a desirable feature of programming languages on resource-constrained platforms. The results in Figure 2.7 appear to support this claim. All non-native C and almost all non-C implementations provide support for managing control flow. (The only exception is found in the work of Motika and von Hanxleden (2015) [126], a pattern whose code is primarily designed as a target for code generators.) Additionally, 86% of the native C cases were able to provide this feature, primarily through macros.

The management of the state of a coroutine's local variables (RQ5c) has also been proposed as a desirable characteristic. Once again, all non-native C and almost all non-C implementations provide support for this feature. None of the native C implementations

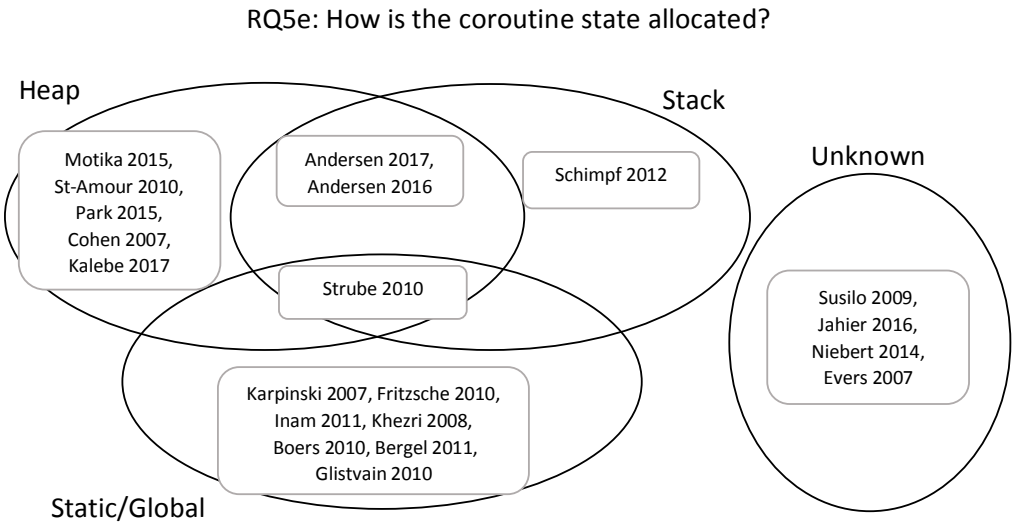


Figure 2.8: RQ5e - How is the coroutine state allocated?

were able to provide it, as a consequence of the language's limitations.

None of the native C implementations and only one of the non-native C implementations were stackful. By contrast, 82% of the non-C implementations were stackful. It could be argued that this split indicates that, while stackfulness is a desirable feature for language designers in general, it is less desirable for C developers. Our interpretation is that, because of the perceived costs of stackfulness in terms of memory and speed, there remains strong support in the C/C++ developer community for stackless coroutines [49].

The allocation of coroutine state is an important feature of the design with regard to its effect on resource-constrained platforms, since it must be controlled carefully if the design is to offer predictable and safe behaviour. Of the 16 implementations where we were able to determine the allocation method, nearly a third used an object or structure to store the state. 44% (7 instances) required that the state be allocated in static (global) memory; 1 used only the stack, and 3 offered flexibility as regards the location.

Five studies ([126, 173, 140, 34, 87]) required that the state be stored on the heap (i.e. in dynamically allocated memory space). Of these, 3 were in languages that required such a strategy (Java, Scheme and Lua) and only two ([34, 87]) used a C-based language (NesC or C++). In the case of [34], each coroutine stack of 256 bytes was allocated on the heap. However, the total number of coroutine stacks required was known in advance, and a safe allocation strategy was therefore feasible. Figure 2.8 summarises these memory strategies.

Given that mainstream C++ programming supports environments where heap memory is generally plentiful, any standard implementation of coroutines in C++ must support dynamic memory allocation for coroutine state storage. However, the special case of resource-constrained platforms, including embedded systems, requires that the developer have the option to use stack memory or global static memory, and that they have full control over which is used on each instantiation. An implementation that supports all three strategies, and allows control over which is used, is therefore desirable.

2.5.2 Research gaps

Focussing specifically on the studies that describe an implementation, we have analysed the issues that were addressed by the research in order to identify gaps, as shown in Table 2.3. Most studies considered the memory and computational cost of the coroutine system, whereas fewer authors addressed interoperability with legacy code. The issue of predictable memory usage by coroutines is particularly important for embedded systems; although 10 of the 21 papers offered a solution, none of these solutions will apply to a C++ native solution which also handles local variable state.

We conclude that a research gap remains with regard to the study of standard C++ as an appropriate language for the development of asynchronous programs on resource-constrained devices.

2.5.3 Repeatability of search results

We found that when the IEEE Xplore database search was repeated 11 months after the original search, the new results were not, as they were expected to be, a superset of the original results. Of the original 144 papers found in October 2017, only 87 (60%) appeared in the search results in September 2018. Further, of the 32 new results, only 16 were papers published since 2015: the other half were published before 2015. We conclude that the search methodology of the IEEE database has changed in the interim, and this raises a question over the use of this database for systematic surveys. This problem was not found for the other on-line databases used.

Table 2.3: Summary of research gaps

Paper	Language was C/C++	Predictable memory usage	Integrates with other language features	Maintainability & readability	Labour cost of implementing the infrastructure	Memory and processing cost of infrastructure	Continued use of legacy code
Cohen et al. 2007		✓✓	✓✓	✓✓	✓✓	✓✓	
Evers et al. 2007		✓				✓✓	
Karpinski et al. 2007		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Kumar et al. 2007	✓✓	✓				✓✓	✓✓
Khezri et al. 2008			✓✓			✓	✓✓
Susilo et al. 2009	✓✓					✓✓	
Boers et al. 2010	✓✓		✓			✓✓	
Fritzsche et al. 2010	✓✓		✓✓				
Glistvain et al. 2010	✓✓	✓✓			✓✓	✓✓	
St-Amour et al. 2010			✓✓	✓✓	✓✓	✓✓	
Strube et al. 2010	✓✓	✓✓	✓				
Bergel et al. 2011		✓✓		✓✓		✓✓	
Inam et al. 2011	✓✓				✓	✓✓	✓
Schimpf 2012	✓✓		✓	✓	✓✓	✓✓	✓
Niebert et al. 2014	✓✓	✓✓				✓✓	
Motika et al. 2015			✓✓	✓		✓✓	
Park et al. 2015		✓✓	✓✓	✓✓		✓✓	✓
Andersen et al. 2016		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Jahier 2016		✓✓	✓✓		✓✓		✓✓
Andersen et al. 2017		✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
Kalebe et al. 2017	✓✓		✓✓	✓✓	✓✓		
Ideal outcome	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓

Key: ✓✓- The issue is considered and resolved. ✓- The issue is addressed but not resolved. Blank – The issue is not present.

2.5.4 Discussion

The majority of selected papers used the C programming language; while the coroutines proposal [81] is for C++, the use of C++ for these projects would not necessarily require

significant programming changes.

Over a quarter of the papers used the Contiki [47] operating system, which provides coroutine support through Protothreads [49], which rely on Duff's device. Given the problems, discussed in Section 2.2.3, that are associated with using this device, this common use of Protothreads indicates a widespread need for the facilities provided by a coroutine-like solution.

More than half (55%) of the studies used 16-bit or 8-bit processors. Support for these platforms on leading C++ compilers is currently limited; this will need to be addressed before C++ coroutines can be applied to the smallest platforms.

As expected [9, 168], code style or simplicity was the leading desired benefit of the language feature implementation. The second most common benefit was a basis for a scheduler: this is not commonly a perceived benefit of coroutines on mainstream platforms, and this difference warrants further study.

The coding of three common use cases - communications, data-flow and sensor readings - present particular difficulties on constrained-resource devices, because these problems require the use of split-phase programming. Each of these problems could be addressed using programming patterns enabled by coroutines: async/await and generator. These patterns would enable a direct programming style that is likely to reduce development effort and the incidence of errors. The high incidence of these use cases in our survey indicate that they represent an important and worthwhile target for further study.

Our survey indicates that multiple studies exist that require a coroutine-based facility for concurrent programming on resource-constrained devices, establishing that a demand exists at this end of the spectrum, not merely on high performance platforms. We noted in Section 2.5.1 that, where the language allowed fine-tuned management of memory allocation, dynamic allocation of memory was avoided for coroutine state and stack. We can conclude that avoiding heap memory is a requirement for the small devices that formed the bulk of the target platforms.

While 82% of non-C implementations are stackful, only one of the C implementations is stackful. We observe that when a language is designed from the ground up to support coroutines, then a stackful implementation is common. On the other hand, such a feature is difficult in C, while preserving both backward compatibility and ac-

ceptable memory usage. We conclude that a stackless implementation is important to C programmers, and this reflects the scarcity of memory resources on the platforms under consideration.

None of the works studied utilize a coroutine implementation for C or C++ that provides managed variable state and that is designed specifically for an event-driven environment on a resource-constrained platform. We therefore conclude that this represents a significant research gap, and that further work towards such an implementation is warranted.

Although this survey found 20 papers that used C and only 2 that used C++, there is evidence that a migration from C to C++ on resource-constrained devices is occurring [9]. Developers may also be motivated to make the switch from C to C++ to gain access to a clean implementation of coroutines to support the async/await and generator patterns and lightweight scheduling, as provided by the proposed C++20 standard [81]. However, this will require language and library support appropriate for resourced constrained devices. Implementers should consider ways to avoid two of the C++ features considered dangerous by Goldthwaite [68]: dynamic memory allocation and exceptions. It would be particularly useful to establish whether the proposed C++ coroutine implementations can offer deterministic memory utilisation, known at compile time: this would make it possible to avoid dynamic memory allocation.

We have seen that various specialized solutions have been applied to the problem of providing direct programming style for split-phase code on embedded systems, including Protothreads, precompilers, language extensions, post-compilation optimization phases and non-portable code libraries. It is clear that, on the one hand, coroutines offer many benefits for software development on these devices but, on the other hand, the implementation is challenging. By contrast, implementing coroutines in C++ on mainstream enterprise systems is relatively straightforward, because there are resources to spare, including memory, operating system facilities and standard libraries. While adapting coroutines for resource constrained devices may be more difficult, it offers greater benefits, because the use cases are such a good fit for embedded systems, including the low-cost, low-power scheduling, communications and sensor management that are often needed by Internet of Things applications.

2.6 Conclusion and further work

2.6.1 Conclusion

This study has analysed the current academic body of work regarding the use of asynchronous programming techniques in embedded systems. We conclude that there exists significant demand for these facilities. We argue that embedded systems must be considered as part of the debate around the standardisation of coroutines in C++. The C++ proposals provide an opportunity to improve the software engineering of embedded systems but only if the language facilities are useful in an extremely resource-constrained environment.

2.6.2 Further work

At the conclusion of this study it was determined that future work should include the following:

- Investigate whether the N4680 proposal can provide deterministic memory use, with full control over the detail of allocation and, if not, what changes would need to be made to the specification. Similarly, test whether the current implementations (Microsoft C++ 14.1 [123] and LLVM 7.0.1 [114]) provide this determinism and control.
- Investigate whether the N4680 proposal and its implementations can work effectively in an event-driven environment on a resource-constrained platform, with and without a real-time operating system.
- Study the memory and performance costs of the current N4680 implementations on resource-constrained platforms with minimal or no operating system support.

Consequently, all these issues were addressed in later chapters of the thesis.

2.7 Appendices

2.7.1 Inclusion and exclusion criteria

The inclusion criteria applied to the articles found by the search are listed in Table 2.4. The corresponding exclusion criteria are listed in Table 2.5.

Table 2.4: Inclusion criteria

Code	Criterion
IC1	The paper contains original research into the application of coroutines on constrained-resource platforms.
IC1a	The application of coroutines must extend to the code on the platform itself, not merely to the simulator of the platform.

Table 2.5: Exclusion criteria

Code	Criterion
EC0	The paper is a duplicate.
EC1	The paper has no digital object identifier (DOI) or International Standard Book Number (ISBN).
EC2	The paper has no abstract.
EC3	The paper was published before 2007.
EC4	The paper is not written in English.
EC5	The complete paper was not available to the reviewers in any form equivalent to the final version.
EC6	The paper is an earlier version of another candidate paper.
EC7	The paper is not a primary study.
EC8	The paper does not fall into any of the selected publication classes.

2.7.2 Hardware classes

The hardware platform classes used for RQ2a are listed in Table 2.6.

Table 2.6: Hardware classes

Class	Data	Code
C0	$<< 10KiB$	$<< 100KiB$
C1	$\approx 10KiB$	$\approx 100KiB$
C2	$\approx 50KiB$	$\approx 250KiB$
$< 1MB$	$< 1000KiB$	N/A
?	Unknown	Unknown

2.7.3 List of papers reviewed

The full list of papers reviewed for the questionnaire is in Table 2.7. The questionnaire results can be found in full in Section 2.7.4.

Table 2.7: Full list of reviewed papers

Code	Author(s)	Year	Title
MOTIKA	Motika & von Hanxleden	2015	Light-weight Synchronous Java (SJL): An approach for programming deterministic reactive systems with Java
SUSILO	Susilo et al.	2009	A miniaturized wireless control platform for robotic capsular endoscopy using advanced pseudokernel approach
ANDERSEN17	Andersen et al.	2017	Enabling synergy in IoT: Platform to service and beyond
YU	Yu et al.	2008	A Survey of Studying on Task Scheduling Mechanism for TinyOS
CLARK	Clark	2009	Powering intelligent instruments with Lua scripting
ELSTS	Elsts et al.	2017	Internet of Things for smart homes: Lessons learned from the SPHERE case study
LOHMANN	Lohmann et al.	2012	The Aspect-Aware Design and Implementation of the CiAO Operating-System Family
JAHIER	Jahier	2016	RDBG: A Reactive Programs Extensible Debugger
ST-AMOUR	St-Amour & Feeley	2010	PICOBIT: A Compact Scheme System for Microcontrollers
NOMAN	Noman et al.	2017	From threads to events: Adapting a lightweight middleware for Contiki OS
PARK	Park et al.	2015	Lua-Based Virtual Machine Platform for Spacecraft On-Board Control Software
ANDERSEN16	Andersen et al.	2016	System Design for a Synergistic, Low Power Mote/BLE Embedded Platform
KARPINSKI	Karpinski & Cahill	2007	High-Level Application Development is Realistic for Wireless Sensor Networks

SCHIMPF	Schimpf	2012	Modified Protothreads for Embedded Systems
FRITZSCHE	Fritzsche & Siemers	2010	Scheduling of time enhanced c (TEC)
ANDALAM	Andalam et al.	2014	A Predictable Framework for Safety-Critical Embedded Systems
HANXLEDEN	von Hanxleden	2009	SyncCharts in C: A Proposal for Light-weight, Deterministic Concurrency
LIU	Liu et al.	2011	Coroutine-Based Synthesis of Efficient Embedded Software From SystemC Models
KUMAR	Kumar et al.	2007	Efficient Software Implementation of Embedded Communication Protocol Controllers Using Asynchronous Software Thread Integration with Time- and Space-efficient Procedure Calls
INAM	Inam et al.	2011	Support for hierarchical scheduling in FreeRTOS
COHEN	Cohen et al.	2007	Using Coroutines for RPC in Sensor Networks
KHEZRI	Khezri et al.	2008	Simplifying Concurrent Programming of Networked Embedded Systems
BOERS	Boers et al.	2010	Developing wireless sensor network applications in a virtual environment
NIEBERT	Niebert & Caralp	2014	Cellular Programming
STRUBE	Strube et al.	2010	Dynamic operator replacement in sensor networks
OLDEWURTEL	Oldewurtel et al.	2009	The RUNES Architecture for Reconfigurable Embedded and Sensor Networks
OLDEWURTEL	Oldewurtel et al.	2009	The RUNES Architecture for Reconfigurable Embedded and Sensor Networks
KUGLER	Kugler et al.	2013	Shimmer, Cooja and Contiki: A new toolset for the simulation of on-node signal processing algorithms
RIEDEL	Riedel et al.	2010	Using web service gateways and code generation for sustainable IoT system development
BERGEL	Bergel et al.	2011	FlowTalk: Language Support for Long-Latency Operations in Embedded Devices

ALVIRA	Alvira & Barton	2013	Small and Inexpensive Single-Board Computer for Autonomous Sailboat Control
DURMAZ	Durmaz et al.	2017	Modelling contiki-based IoT systems
EVERS	Evers et al.	2007	SensorScheme: Supply chain management automation using Wireless Sensor Networks
JAASKELAINEN	Jääskeläinen et al.	2008	Reducing Context Switch Overhead with Compiler-Assisted Threading

2.7.4 Questionnaire content

The full results of the questionnaire are listed below in Table 2.8 (RQ1 & RQ2), Table 2.9 (RQ3 & RQ4) and Table 2.10 (RQ5).

Table 2.8: Questionnaire results - RQ1 Software Platform & RQ2 Hardware Platform

Code	RQ1a	RQ1b Coroutine implementation				RQ1c	RQ2a	RQ2b	RQ2c	RQ2d	RQ2e	RQ2f
	Programming Language	Native	Macros	Library	Other	Operating system	Hardware class	Non-volatile memory (kb)	RAM (kb)	Processor family	CPU bits	Instruction set
MOTIKA	Java	✓			13	JVM	C2	256	64	ARM7	32	ARMv4T
SUSILO	C	✓				Unique	C1	128	8	8051	8	8051
ANDERSEN17	Lua	✓		✓	1	TinyOS	C2	512	64	ARM M4	32	Armv7E-M
YU	nesC	✓			12	TinyOS	C0	132	4	megaAVR	8	AtAVR
CLARK	Lua	✓				Not specified	$< 1MB$	512	256	Moto 68332	32	Moto 68020
ELSTS	C	✓				Contiki	C1	128	20	ARM M3	32	ARMv7-M
LOHMANN	AC++					Generated	N/A	N/A	N/A	TriCore	32	TriCore
JAHIER	Oca				9,14	Generated	N/A	N/A	N/A	N/A	0	N/A
ST-AMOUR	Sch	✓				PICOBIT	C0	32.25	1.5	PIC18	16	PIC17
NOMAN	C	✓	✓			Contiki	C1	256	16	MSP430	16	MSP430
PARK	Lua	✓			7	RTEMS 4.10	$< 1MB$	Unknown	512	SPARC V8	32	SPARC V8e
ANDERSEN16	Lua	✓		✓		TinyOS	C2	512	64	ARM M4	32	Armv7E-M
KARPINSKI	SOL				14	TinyOS	C0	132	4	megaAVR	8	AtAVR
SCHIMPF	C	✓				Unique	C0	33	2	megaAVR	8	AtAVR
FRITZSCHE	C				11	Unique	N/A	N/A	N/A	N/A	0	N/A
ANDALAM	C	✓	✓	✓		Unique	N/A	N/A	N/A	Microblaze	32	Microblaze
HANXLEDEN	C	✓	✓	✓		Unique	N/A	N/A	N/A	N/A	0	N/A

LIU	C	✓	✓		Portable	N/A	N/A	N/A	N/A	0	N/A
KUMAR	C			2	Generated	C0	132	4	megaAVR	8	AtAVR
INAM	C	✓	✓		FreeRTOS	C2	512	64	AVR32	32	AtAVR32
COHEN	nesC	✓		12	TinyOS	C0	132	4	megaAVR	8	AtAVR
KHEZRI	nesC	✓		10	TinyOS	C0	N/A	N/A	megaAVR	8	AtAVR
BOERS	C	✓	✓		PicOS	C0	64	4	eCog1	16	eCog16
NIEBERT	C			4	Not specified	C0	?	4	N/A	0	N/A
STRUBE	C	✓	✓		Contiki	C0	64	10	MSP430	16	MSP430
OLDEWURTEL	C	✓	✓		FreeRTOS	C2	512	32	ARM7	32	ARMv4T
OLDEWURTEL	C				Contiki	C1	64	10	MSP430	16	MSP430
KUGLER	C	✓	✓		Contiki	C1	48	10	MSP430	16	MSP430
RIEDEL	C	✓	✓		Contiki	C2	192	96	JN5139	32	JN5139
BERGEL	FT	✓		3	TinyOS	C0	132	4	megaAVR	8	AtAVR
ALVIRA	C	✓	✓		Contiki	C2	208	96	ARM7	32	ARMv4T
DURMAZ	C	✓	✓		Contiki	C0	N/A	N/A	N/A	0	N/A
EVERS	Sch			9	TinyOS;Contiki	C1	48	10	MSP430	16	MSP430
JAASKELAINEN	C			2	Unique	?	N/A	N/A	N/A	0	N/A

Abbreviations:

RQ1a (Language): AC++:AspectC++/C++, FT:FlowTalk, Oca:Ocaml, Sch:Scheme

RQ1b (Other): 1:Closures, 2:Compiler phase, 3:Continuations, 4:Language extensions, 7:Manipulation of hosting API (in C), 9:New language, 10:Non-portable assembly language, 11:Precompiler, 12:setjmp/longjmp, 13:State machine driven by switch statement, 14:Transpiler

RQ2a (Hardware class): See Table 2.6 for definitions.

RQ2d (Processor family): ARM M3:ARM Cortex-M3, ARM M4:ARM Cortex-M4, Moto 68332:Motorola 68332

RQ2f (Instruction set): AtAVR:Atmel AVR, AtAVR32:Atmel AVR32, eCog16:eCog 16-bit, Moto 68020:Motorola 68020

Table 2.9: Questionnaire results - RQ3 & 4 Use case & Intended Benefits

Code	RQ3: Use cases	RQ4: Intended benefits
MOTIKA	Producer-Consumer;	Deterministic cooperative concurrency;Efficient concurrency in Java
SUSILO	Cooperative multitasking;Producer-consumer	Low CPU and memory overhead for hard RT system
ANDERSEN17	Asynchronous communications;	Programming ergonomics
YU	Sensor readings	Sequential coding style
CLARK	Multitasking	Run multiple scripts
ELSTS	Asynchronous communications;	Simplicity of coding
LOHMANN	Continuations;Interrupt handlers	Not specified
JAHIER	To execute debugger and debuggee code side by side as coroutines	An efficient way to implement debugger coroutining by using continuations
ST-AMOUR	Multithreading	Implement continuations
NOMAN	Communications	Event based interoperability middleware
PARK	Cooperative multitasking with preemptive override	Mapping of On-Board Control Procedures (OBCP) to Lua coroutines
ANDERSEN16	i2c request	”Reduce application complexity by allowing for pseudo-synchronous programming with coroutines”
KARPINSKI	Sensor readings; Asynchronous communications;	”Hides the split-phase program execution scheme and provides programmers with a fine-grained concurrency model to structure event handling”
SCHIMPF	Concurrent programming	Prioritised scheduling with protothreads
FRITZSCHE	Emulating preemption using deconstruction of tasks into coroutines	Scheduling in a real-time system

ANDALAM	Producer-Consumer;Synchronous communications;motor control;multithreading	Performant deterministic concurrency in C;Simplify WCET analysis
HANXLEDEN	Producer-Consumer-Observer;Asynchronous communications	Embed deterministic reactive flow into C
LIU	Asynchronous communications	Operating system portability (via pure C protothreads);Lower memory consumption;Faster execution
KUMAR	Asynchronous communications;Implementing network protocol controllers in software	Lower memory consumption;Faster execution;Efficient use of idle time
INAM	Not specified	Not specified
COHEN	RPC;Communications on WSN	Sequential coding style
KHEZRI	Lengthy tasks e.g. calculation	Resumable tasks in TinyOS, replacing fragmented tasks
BOERS	Communications	State machine implementation with strong similarities between deliverable and simulated
NIEBERT	Multitasking	Simple concurrent programming
STRUBE	Sensor readings	Serialisable coroutine state
OLDEWURTEL	Routing;data aggregation;DSP	Implementation of portable middleware
OLDEWURTEL	Sensor readings	Implementation of portable middleware
KUGLER	Streaming;data preprocessing;producer-consumer	Easy simulation of of wearable nodes
RIEDEL	Concurrent programming	Execute automata for web services at the same time as communications channels
BERGEL	Sensor readings;actuator setting	Sequential coding style
ALVIRA	Concurrent I/O	Concurrent control of multiple actuators and of communications
DURMAZ	Concurrent cooperative tasks:sensor reading, aggregation, communications	Simplification of event-driven programming by reducing explicit state machines

EVERS	Blocking I/O on event-driven platform	To implement multiple threads of control; to enable blocking I/O calls
JAASKELAINEN	Concurrent programming	Optimised task-switching

Table 2.10: Questionnaire results - RQ5 What is the API of the coroutine?

Code	RQ5a: Is there an implementation of coroutines?	RQ5b: Is the control flow managed on behalf of the developer?	RQ5c: Is the state of local variables automatically managed?	RQ5d: Is the coroutine implementation stackless or stackfull?	RQ5e: How is the coroutine state allocated?
MOTIKA	✓			Stackless	Data member;Heap
SUSILO	✓			Stackless	Unknown
ANDERSEN17	✓	✓	✓	Stackful	Data member;Heap or stack
JAHIER	✓	✓	✓	Unknown	Unknown
ST-AMOUR	✓	✓	✓	Stackful	Heap
PARK	✓	✓	✓	Stackful	Data member;Heap
ANDERSEN16	✓	✓	✓	Stackful	Data member;Heap or stack
KARPINSKI	✓	✓	✓	Stackless	Data member;Static
SCHIMPF	✓	✓		Stackless	Stack
FRITZSCHE	✓	✓	✓	Stackless	Static
KUMAR	✓	✓	✓	Stackless	N/A
INAM	✓	✓		Stackless	Static
COHEN	✓	✓	✓	Stackful	Heap
KHEZRI	✓	✓	✓	Stackful	Static
BOERS	✓	✓		Stackless	Static
NIEBERT	✓	✓	✓	Unknown	Unknown
STRUBE	✓	✓		Stackless	Heap, stack or static
BERGEL	✓	✓	✓	Stackful	Static
EVERS	✓	✓	✓	Stackful	Unknown

Chapter 3

C++20 Coroutines on Microcontrollers

This chapter is an expanded and reformatted version of the following paper in the IEEE Journal *IEEE Embedded Systems Letters*.

> B. Belson, W. Xiang, J. Holdsworth and B. Philippa, "C++20 Coroutines on Microcontrollers—What We Learned," in *IEEE Embedded Systems Letters*, vol. 13, no. 1, pp. 9-12, March 2021, doi: 10.1109/LES.2020.2973397.

The paper was published in February 2020, after the March 2019 final agreement regarding the inclusion of coroutines in C++20[1], but before the May 2020 release of coroutine support in GCC[2]. The research underlying the paper was carried out between July 2018 and May 2019 and the paper was written between April 2019 and September 2019. The paper therefore offered an opportunity to comment on issues that might affect the implementations of coroutines in C++ tool-chains, but not to influence the standard itself.

Appendices have been added that provide more detail with regard to (i) source code; (ii) the methodology applied to the support library development; (iii) experimental equipment; and (iv) data pipeline.

[1] https://github.com/cplusplus/draft/blob/main/papers/n4811.md
[2] https://gcc.gnu.org/gcc-10/changes.html

Chapter Abstract

Coroutines were added to C++ as part of the C++20 standard. Coroutines provide native language support for asynchronous operations. This study evaluates the C++ coroutine specification from the perspective of embedded systems developers. We find that the proposed language features are generally beneficial but that memory management of the coroutine state needs to be improved. Our experiments on an ARM Cortex-M4 microcontroller evaluate the time and memory costs of coroutines in comparison with alternatives, and we show that context switching with coroutines is significantly faster than with thread-based real time operating systems. Furthermore, we analysed the impact of these language features on prototypical IoT sensor software. We find that the proposed language enhancements potentially bring significant benefits to programming in C++ for embedded computers, but that the implementation imposes constraints that may prevent its widespread acceptance among the embedded development community.

3.1 Introduction

Coroutines are a programming language mechanism to enable a function to be suspended and resumed. They assist with asynchronous programming, for example, where the program is waiting on an external event. The programmer can simplify their code by combining a request and response into the same coroutine. Coroutines were originally proposed in 1963 [36] but did not see widespread adoption in the decades that followed. However, recent times have brought about a revival in interest [40, 125, 42, 144]. Coroutine features are now present in many mainstream programming languages including C# [136], Lua [41, 7], JavaScript and Python.

C and C++, the dominant languages for embedded systems programming, have until recently lacked native support for cooperative multitasking. Many attempts have been made to use coroutines in C and C++ for cooperative multitasking or as a lightweight thread replacement [17]. However, the majority of these have been based on preprocessor macros and Duff's device[3], including Protothreads [49] and FreeRTOS [15]. These implementations do not manage the state of local variables during coroutine sus-

[3]https://www.lysator.liu.se/c/duffs-device.html

pension, and Duff's device is arguably inappropriate for production-quality code. Nevertheless, the fact that these methods have been used so often is an indicator of how useful coroutines could be if they had first-class language support.

Coroutines were added to C++ in the C++20 standard [82]. Thus far, most analysis of C++ coroutines has focussed on high performance computing [85]. Relatively little attention has been paid to embedded systems [17], despite coroutine features from other languages being used in this space [45, 156].

We argue that embedded software is becoming more important with the emergence of the Internet of Things (IoT). Embedded devices that are connected to the internet must face a wider range of security threats, and methods to reduce the occurance of bugs are important if IoT software is to be widely accepted. IoT programmers are accustomed to using asynchronous language features on server platforms; they may expect these features to be present also on the sensor platforms that communicate with these servers. The new C++ language features provide such an opportunity.

Our three main contributions are as follows. Firstly, we analyzed the appropriateness of the C++ coroutines Technical Specification for embedded systems, which is an application space that does not seem to have previously been considered, and we experimentally assessed the viability of the proposal by implementing it on an ARM Cortex-M4 microcontroller with 128 kb of memory. Secondly, we conducted benchmarks to assess the time and memory costs of coroutine features in comparison to other approaches. Finally, we analyzed the effect of coroutines on the complexity and length of realistic programs by building a prototypical IoT device, both with and without coroutines. We find that coroutines are viable on memory-constrained embedded platforms, although there are some issues with the design of the Technical Specification that might affect their acceptance within the community.

3.2 Analysis of the appropriateness of the coroutine standard for microcontrollers

3.2.1 New Language Features

A coroutine can be created in C++20 code by invoking a new keyword *co_await*, as demonstrated in Listing 3.1. A function *someTask()* calls *result = co_await adc_read();*.

```
1  resumable someTask() {
2    co_await adc_calibrate();
3    uint8_t result = co_await adc_read();
4    process_data(result);
5  }
```

Listing 3.1: Using co_await to read a sensor in C++

adc_read() immediately returns an awaitable object, such as a *future<>*, whose payload will be the output data. This triggers *someTask()* to suspend itself. Another execution context, such as a thread or an interrupt service routine (ISR), uses the awaitable object to signal the scheduler that it is complete; the scheduler resumes *someTask()* at the point where it was suspended, with its state, including parameters and local variables, fully restored.

Because *someTask()* used the *co_await* facility, the standard recognises it as a coroutine: the compiler generates the code for a coroutine object, exposed to the application opaquely as a typeless handle. The code includes a finite state machine (FSM) which implements the split-phase code. *someTask()* is replaced by a number of small functions, one for each state: their invocation, their shared data and the machine state is managed by the coroutine, in cooperation with the scheduler.

These features have been implemented in three C++ compilers: Microsoft C++ (since 2016), LLVM/clang (2017) and EDG (2015); a partial implementation is available as part of GCC (as of February 2019) [172]. The Microsoft implementation has been used by "thousands of software developers in various companies" and "software built using coroutines has been deployed on more than 400 million devices." [132]

There has been considerable discussion of the proposed standard: it has not been universally accepted in the C++ language developer community [154, 172], and an incremental approach that incorporates other proposals is now likely [131].

We note that coroutines, as a mechanism to control the flow of execution, are often described as a way to achieve concurrent programming; however, they are better considered as "a first step towards concurrency" [25]. In this chapter we follow the usage of the C++20 standard; more precise terminology can be found in [25].

3.2.2 Coroutine Stack Frame

While a coroutine is suspended, its execution state, local variables and parameters are all stored in a data frame assigned by the compiler, the coroutine stack frame (CSF). The proposed standard specifies that the CSF is, by default, allocated dynamically on the heap.

The standard provides for an alternative allocation policy, which requires modification of the library classes, and is not appropriate for end-users - application developers - since it adds significant complexity to the use of coroutines, and repeatedly exposes implementation details.

Furthermore, the size of the memory required for the CSF is not known at compile time, since it may be amended by later optimiser passes. The compiler does not provide a mechanism to access the CSF size to use for static memory allocation, as illustrated in Figs. 3.1 and 3.2.

3.2.3 Standard Library

We chose the clang compiler for this study due to its ARM backend and the relative maturity of its coroutine implementation. However, the existing header files were not suitable for memory-constrained microcontrollers. Specifically, the C++20 coroutines technical specification [82] requires a set of template classes, *std::coroutine_** . Widely used implementations of these classes depend upon C++ features that are often avoided in embedded programming, such as exceptions and dynamic memory allocation [68]. Consequently, the existing libraries do not demonstrate idiomatic embedded code and were deemed unsuitable for our experiments.

Our analysis therefore required that we create a new C++ coroutine standard library targeted at resource-constrained embedded platforms. It implements all the required *std::coroutine_** templates, a specialised version of *future<>* without dependencies, and also some features specifically created for microcontrollers. For example, a specialised class *split_phase_event_t* provides the infrastructure for the split-phase operations such as triggering a peripheral and responding to its interrupt. Helper classes were also created to ease and standardise the programming of GPIO ports, the I^2C bus, serial ports and timers. Our C++ library is published online under the MIT license[4].

[4] https://github.com/bbelson2/coro-mc-wwl-code

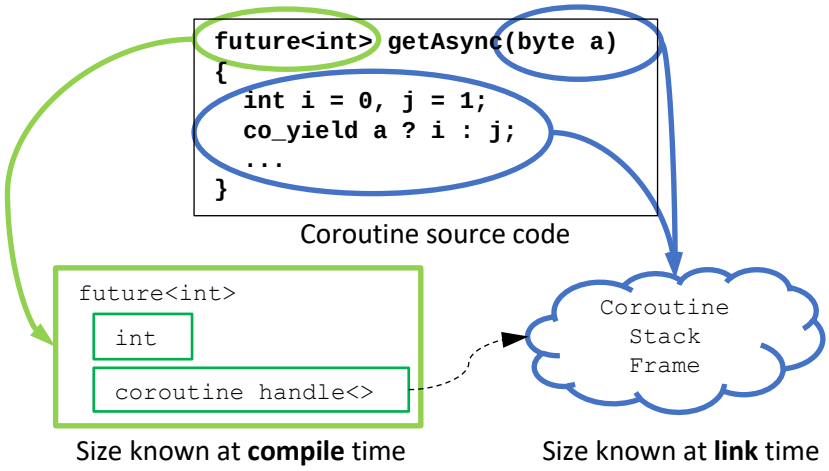


Figure 3.1: Memory layout for a resumable function.

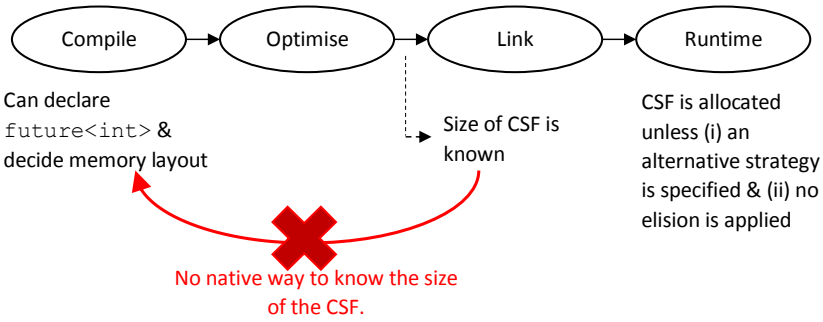


Figure 3.2: Coroutine stack frame workflow.

3.3 Experimental results

To examine the implementation, we built two sets of test applications for a microcontroller:

1. a microbenchmark to measure the time and memory costs of context switching; and

2. a simple application to read multiple sensors and process the data.

Each application ran on the Freescale FRDM-K22F development board, with an ARM Cortex-M4 32bit 120 MHz microcontroller. We used the Eclipse-hosted Kinetis Design Studio, with the LLVM/clang tool chain (version 8.0.0). Code and memory sizes were measured with `arm-none-eabi-size`.

3.3.1 Context Switching Microbenchmark

The time cost of switching among tasks was measured with a microbenchmark that switched the voltage of a GPIO pin using two tasks: one task to set the pin high, and the second to set it low. The tasks were run in a tight loop and the scheduler switched tasks as quickly as possible. The GPIO voltage was measured using a Rohde & Schwarz HMO2024 oscilloscope, capturing 4 GSa/s. We took advantage of the oscilloscope's built-in capability for measuring the frequency of square waves. The test was repeated using two different development boards and we found no observable change in the results.

To adapt the C++20 coroutine into a cooperative multitasking scheduler comparable to that of an embedded real-time operating system, a scheduler class and a task class were created. The scheduler class was responsible for selecting the next task, guided by task priority and task readiness, and then instructing the task instance to resume. The task class was a simple wrapper around a coroutine instance, and directly invoked the coroutine's `coroutine_handle<>::resume()` method. As appropriate for a cooperative system, each coroutine was responsible for pausing, by invoking `co_await`.

The coroutine used in this microbenchmark was very simple, with the following steps:

1. `co_await suspend_always{}; // Immediately yield`
2. `while (true) { // Loop forever`
3. `    // This is where the coroutine will be resumed...`
4. `    toggle_gpio_pin();`
5. `    co_await suspend_always{}; // Yield`
6. `}`

It was observed that the coroutine approach took about 0.209 μs for each context switch, which is more than an order of magnitude faster than the 2.504 μs cost using the real-time operating system, FreeRTOS [15], and 2.589 μs under MQX Lite [135], as shown in Figure 3.3. While the coroutine context switch is slower than that of Protothreads [49], we consider it to be preferable: it preserves local variables, it is more readable and it does not damage the language by restricting `break` statements.

The code and data sizes of the various minimal applications are compared in Figure 3.4. There is a small (8 byte) overhead for coroutine data compared to the minimal

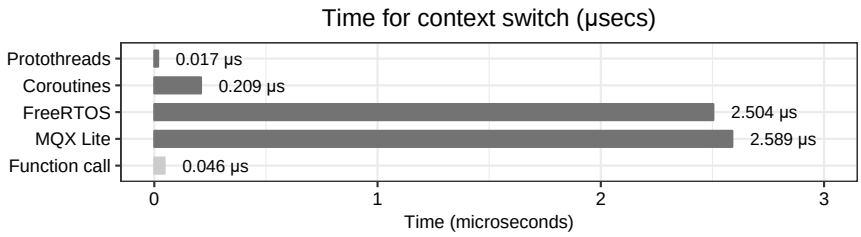


Figure 3.3: Time cost of context switching microbenchmark. We emphasize that Protothreads does not provide a genuine context switch since it does not restore local variables. Observe that coroutines are significantly faster than the thread-based mechanisms of FreeRTOS and MQX Lite. The time cost of a simple function call (which adds two integers) is added for comparison.

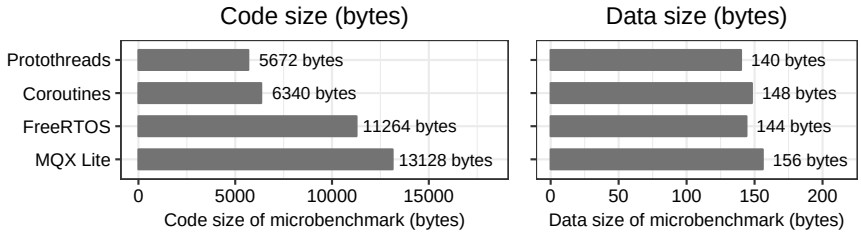


Figure 3.4: Memory cost of context switching microbenchmark.

Protothread implementation. The coroutine application is about 12% larger than the Protothread version, and is 44% and 52% smaller than the RTOS versions.

3.3.2 Memory Costs

The cost of using coroutines, in terms of code size, depends on many variable factors which impact on the optimisation phases. Our simple tests showed that there was a fixed cost of 704 bytes caused by the inclusion of any coroutine – the infrastructure cost – and costs associated with the creation of each resumable coroutine (4 bytes) and each invocation (56 bytes).

3.3.3 Ergonomically Efficient Code

We analyzed the ergonomic benefits of a coroutine style by building a sample application. We used the same scheduler and task classes as the context switching benchmark in Section 3.3.1. For the application programmer, it became very quick and simple to program asynchronous operations; the code could be written in a continuous, self-contained style. The impact of coroutines on code size is shown in Table 3.1 in terms of source lines of code (SLOC). For our sample application (which used the analog-digital converter and the I^2C bus), the coroutine source code was shorter than the code for the

Table 3.1: Source lines of code for asynchronous tasks

Platform	Library	Application	Total
Finite state machine	-	363	363
Coroutines	196	60	256

finite state machine (256 SLOC vs 363). Excluding the generic library code from the count, the difference was more pronounced: the coroutine version was 75% shorter.

A typical usage was shown in Listing 3.1. The three asynchronous operations are listed sequentially in a single unit of code; the *co_await* keyword – in cooperation with the scheduler – provides all the asynchronous behaviour, and the developer is not required to create a finite state machine, or to use the continuation-passing style common in Javascript.

In our opinion, the coroutine version was simpler to read and therefore easier to maintain, compared to the traditional finite state machine pattern or a continuation-passing style.

3.3.4 Zero- and Negative-cost Abstractions

A common coding practise with simple low-latency asynchronous operations is to run code in a tight wait loop: a "busy wait". For example, when writing a byte via a serial connection, it is simpler to wait (very briefly) for the local serial port to acknowledge the latest byte, rather than add the complexity of a split-phase operation. Any higher level of efficiency that might be provided by a split-phase operation *without coroutines* would require extra developer effort in communicating with an ISR, coding a finite state machine, maintaining the FSM's state and values and coordinating the main routine's suspension and resumption with the scheduler.

Using a blocking method and a coroutine, this higher efficiency can be achieved without any added code complexity for the application programmer, assuming the use of appropriate library code that blocks only when necessary, and does so transparently. This is a win-win outcome: it avoids the trade-off between ease-of-programming and efficiency and results in simple source code with efficient performance.

3.4 Discussion

3.4.1 End-user experience

Our experience was that typical asynchronous code patterns for microcontrollers can be written easily and intelligibly in C++ using the new language features. We consider that the experience for the application developer is positive: complex code patterns can be expressed simply, concisely and clearly.

3.4.2 Performance cost

The performance cost is extremely low, on the order of 0.21 μs on a 120MHz device, which is 12 times as fast as a context switch using a RTOS. In some cases the overall impact on performance may be positive, because the compiler is able to optimise code patterns that are not detected when they are presented as a traditional finite state machine.

There is a single code size cost of $\approx$700 bytes for the use of any coroutines, but the incremental cost of creating and calling each coroutine is small: 4 bytes and 56 bytes, respectively.

3.4.3 Library support

The unique execution context of microcontrollers, particularly the demands of ISRs, requires specialised classes for low-level asynchronous support. Further, it is frequently not acceptable to use many standard C++ library features, including dynamic (heap) memory allocation, exceptions, strings and standard containers. This means that a support library for use on an embedded platform must be based on a specialised version of the standard library.

3.4.4 Memory allocation

The ISO proposal does not facilitate fine-grained control of the memory used for the coroutine stack frame: until such control is provided, takeup of these features among the embedded programming community is likely to be low. Some suggestions for addressing this problem have already been made in the ISO forum, but none have yet addressed the context of embedded platforms specifically. There are significant language and tool

Table 3.2: Impact of platform considerations

Platform	CPU	Bit width	RAM (kB)	Flash (kB)	Compiler support
FRDM-K22F	ARM Cortex-M4	32	128	512	GCC, LLVM
Arduino Uno	ATMega328P	8	2	32	GCC‡
PIC18F24Q24	PIC18	8	4	32	–
ESP8266	Tensilica L106	32	160	4096	–
ESP32	Xtensa LX6 x2	32	520	4096	GCC†
Raspberry Pi Pico	ARM Cortex-M0+ x2	32	264	2048	GCC, LLVM†

†: Experimental support only.

‡: No coroutine elision support means that all stack frame allocation is on the heap.

chain problems that prevent a simple and elegant solution to this problem under the current proposal.

Our current workaround was a "two stage" compilation. The first compilation finds the size of all coroutine stack frames, which are manually copied into a header file for use in the second compilation. The second compilation was able to statically allocate the required memory. We anticipate that such a mechanism could be built into future compilers to provide a standard process for static allocation of coroutine state.

3.4.5 Platform considerations

The platform selected for this study - the Freescale FRDM-K22F development board - provides a relatively powerful resource-constrained device, with an ARM Cortex-M4 32bit microcontroller (which supports the ARMv7-M Thumb instruction set) along with 128 kB of RAM and 512 kB of flash memory. Table 3.2 shows the platform characteristics in the context of other microcontroller platforms used in edge and embedded systems.

When we consider that – in our tests – the cost of using coroutines in terms of added code size is 704 bytes plus 56 bytes per invocation of a coroutine, it is clear that popular 8-bit platforms such as the Arduino Uno or the PIC18 family are not always appropriate targets, with a limit of only 32 kB of code. The advantage of simpler source code may be outweighed by the absorption of at least 2% of the available code size. Furthermore, the amount of run-time memory used by each running (or suspended) coroutine – 148 bytes in our tests – is very significant in the context of the 2048 or 4096 byte total limit for the Arduino Uno and the PIC18 respectively. Finally, the availability of stable compiler

support for C++20 features is an absolute constraint that would – at this time – prevent the use of coroutines on the Arduino Uno, the ESP8266 and the PIC18 family.

3.5 Conclusion

In conclusion, we find that the new C++20 coroutine features assist the development of asynchronous code for embedded platforms: the resulting code has low overhead and encourages more elegant and transparent design. However, for these constrained-resource devices, the memory allocation methods do not yet offer sufficient control to satisfy the needs of the embedded programming community.

3.6 Appendix I: Source code

The public Github repo containing the source code used in this study can be found at https://github.com/bbelson2/coro-mc-wwl-code.

3.7 Appendix II: Development problems and process

3.7.1 Overview

This section describes the coroutine support runtime library created for the projects in this study. It contains a summary of the library development phases and detailed discussions of problems encountered with memory allocation.

3.7.2 Objectives

The following characteristics were chosen as development goals for the library.

- Minimal dependencies on Standard Template Library (STL).
- Ideally, no dependencies on STL.
- No use of heap memory.
- No use of exceptions.
- Transparency of coding techniques, particularly from the point-of-view of C programmers (as opposed to experienced C++ programmers).

3.7.3 First iteration

The code for the first iteration can be found in the subfolder `archive/iteration1` of the on-line repository.

3.7.3.1 Contents

The first version of the library included the files and classes listed in Table 3.3. The application project `k22fawait1` makes use of these classes to build hardware abstraction layers (HALs) as described in Table 3.4. For the application project, application-level test classes based on the HAL code were created, as listed in Table 3.5:

Table 3.3: File list for first iteration

File	Classes	Description
experimental/resumable	coroutine_traits<>	Coroutine contract from N4680 & N4736
	coroutine_handle<>	
	suspend_never	
	suspend_always	
core_crit_sec.h	mutex_t	Critical section adapter class
core_resumable.h	resumable<>	Base resumable type, used for tasks
core_future.h	future_t<>	Futures with:
	promise_t<>	(i) embedded awaitable contract &
	awaitable_state<>	(ii) shared state on heap
core_scheduler.h	task_t	Simple scheduler
	scheduler_t	

Table 3.4: First iteration hardware abstraction layer

File	Content
api_adc.h/.cpp	Asynchronously read an analog to digital converter channel
api_i2c.h/.cpp	Asynchronously read and write to a I2C bus (for use with the development board's accelerometer)
api_timer.h/.cpp	Asynchronous timer event stream

Table 3.5: First iteration application classes

File	Content
task_adc.cpp	Asynchronously read a pair of analog to digital converter channels (x and y coordinates from a joystick)
task_i2c.cpp	Asynchronously read and write to a I2C bus, thereby reading the development board's accelerometer
task_timer.cpp	Asynchronous timer event stream

3.7.3.2 Benefits

The library delivered significant benefits both to the application programmer and the HAL library writer.

For example, the ADC task could include efficient and transparent asynchronous code such as that shown in Listing 3.2. The code is ordered and is not fragmented: the intent of the code is very clear compared to alternative methods such as a finite state machine (FSM).

```
co_await start_adc(ADC_CHANNEL_X);
co_await start_adc(ADC_CHANNEL_Y);

for (;;) {
  auto x = co_await read_adc(ADC_CHANNEL_X);
  auto y = co_await read_adc(ADC_CHANNEL_Y);

  // Use x, y

  co_await wait_on_ticks(10);
}
```

Listing 3.2: Using `co_await` to write C++ code in a direct style without fragmentation

3.7.3.3 Problems

There are a number of problems with the first iteration.

- The split phase event class `split_phase_event_t` makes use of lambda expressions. The use of lambdas may be considered to fail the transparency objective with regards to C programmers.
- The split phase event class `split_phase_event_t` makes use of classes in STL's `<functional>`, to store lambda expressions for deferred operations. (These lambdas cannot be stored as function pointers because they use captures.) In a future iteration, therefore, we will avoid lambdas and instead use void function pointers; in place of captures, we will use global static data.
- `future_t` and its related classes use dynamic memory allocation. Because `future_t` and `promise_t` maintain a shared `awaitable_state_t` via a usage count, the memory is dynamically allocated (through a smart pointer). We will avoid this in the second iteration by by using global static data storage for promise and state.
- The scheduler uses the STL `stack<>` container to hold the current set of blocked coroutines for a task.
- The I2C api is too complex (see `archive/iteration1/Sources/api_i2c.cpp`); it contains too much secondary level code, which should be split out into an implementation layer. Low-level calls which read and write byte arrays should be

```
future_t<byte> read_i2c(uint8_t slave_address,
    uint8_t reg, uint8_t* data, word len) {
  byte rc = I2C_SelectSlave(slave_address);
  if (rc == 0) {
    rc = co_await I2C_SendChar_async(reg);
  }
  if (rc == 0) {
    word recv;
    rc = co_await I2C_RecvBlock_async(data, len, &recv);
  }
  I2C_SendStop();
  co_return rc;
}
```

Listing 3.3: C++ code demonstrating composition of coroutines

implemented as asynchronous primitives delivered as coroutines - each matching a synchronous primitive. The current read and write calls should be coroutines composed of these primitives.

3.7.4 Second iteration

The code for the second iteration can be found in the subfolder `archive/iteration2` of the on-line repository.

3.7.4.1 Problem

Composition of coroutines (such as the code in Listing 3.3) fails with the initial `future_t<>` implementation. The classes `future_t<>` and `promise_t<>` contain both the core code of the classes – including the constructors – and the methods of the awaitable contract.

Thus the coroutine in Listing 3.3 (which creates a `future_t<>` via `future_t<>::promise_type`) attempts to call a constructor for `future_t<>` with all four of the arguments.

3.7.4.2 Objective

- Make composition of coroutines work correctly by separating the awaitable classes' core implementation and their awaitable contracts. In particular, the types

```
namespace std { namespace experimental {
template<class T,
class... Args>
struct coroutine_traits<future_t<T>, Args...>
{
  struct promise_type {
    promise_t<T> _promise;
    ...
  };
};
```

Listing 3.4: Redesign of `future_t` class

`future_t::promise_type` and `promise_t<>` must be separated.

3.7.4.3 Description

The `future_t<>` awaitable contract was moved to the `std::experimental::coroutine_traits<>` extension class, as summarised in Listing 3.4.

3.7.4.4 Benefits

- Composition of coroutines is now functional.

3.7.4.5 Observation

The version 1 code appeared correct according to the standard; however, it failed to compile as desired. This solution used an opaque and unobvious work-around. It is arguable that the standard – in this case – is too complex to implement in a straightforward fashion.

3.7.5 Third iteration

3.7.5.1 Objectives

- Remove heap dependency of `future_t<>` (the shared state).
- Remove remaining STL class `std::stack<>`, used for the list of blocked coroutines.

3.7.5.2 Discussion

A linked list for coroutines

In order to remove `stack<coroutine_handle<>>` (or another dynamic container), we can change the architecture, by adding an active coroutine stack to `task_t`, implemented as a linked list. All awaitables must now inherit from a class which links to the next awaitable.

But which object can become a member of the linked list? The `future_t<>` is not eligible, because it is constrained to have move semantics but no copy semantics. Nor is `promise_t<>`, because it is not always created explicitly: the `future_t<>::promise_type` creates the `promise_t<>`. It appears that the best – perhaps the only – candidate is the shared state class.

A heap-free shared state

The shared state is currently created within a counted shared smart pointer, `counted_ptr<state_t>`. The `state_t` is a template parameter of `future_t<>` (and, in parallel, of `promise_t<T, state_t>`).

We can make this arrangement less inflexible by promoting the shared state holder (e.g. `counted_ptr<>`) to become a template argument. This will allow us to use a weak pointer instead of a shared pointer if we wish. Such an arrangement would allow the state to be embedded within the `promise_t<>`, when the promise is static in scope and lifetime.

3.7.5.3 Two creation patterns for `future_t<>`

1. Explicitly created in a split-phase coroutine, using `promise_t::get_future()`.
2. Implicitly created during a `co_return` from a coroutine, via `future_t<>::promise_type::get_return_object()`.

First we investigate these in detail, then examine their suitability under the new state architecture.

Pattern 1: Explicit construction during split-phase

1. During coroutine, `promise_t<>` is created as a local variable, with default value.

2. promise_t<>'s state is passed to the deferred lambda as a capture.

3. After immediate action, promise_t<>::get_future() is called explicitly by coroutine just before exit. (Deferred action may or may not have taken place by now.)

Pattern 2: Created during co_return

1. During coroutine ramp, promise_type is instantiated. Nested structure promise_t<> is constructed within promise_type. There are no constructor parameters for promise_t<>. As a result, the awaitable state's data is initialised to the default value: T().

2. During co_return, promise_type::return_value(rc) is called, which sets the value of the promise_t<>'s state, via _promise.set_value(rc).

3. Next, promise_type::get_return_object() calls _promise.get_future() which creates the future_t<> using the (now resolved) state as the constructor's parameter.

3.7.5.4 New contract for shared state pointer

We introduce a new contract for the shared state, shared_state_ptr_t:

```
static shared_state_ptr_t shared_state_ptr_t::make_ptr(...);
typedef [the contained state type] state_type;
shared_state_ptr_t& operator=(const shared_state_ptr_t& cp);
shared_state_ptr_t& operator=(shared_state_ptr_t&& cp);
```

These allow the future_t and the promise_t to (i) create a new instance of the shared state pointer, passing the constructor parameters via perfect forwarding, (ii) access the shared data and (iii/iv) copy the pointer into a member variable.

```
template <typename T,
  typename shared_state_ptr_t = counted_ptr<awaitable_state<T>>>
struct future_t { ... };
```

The intent of this approach was to retain a single future_t<> class which could be used with both static and dynamic allocation. The dynamic case would continue to be implemented using the counted_ptr<> thus:

```
future_t<word, counted_ptr<awaitable_state<word>>> f1;
future_t<word> f1; // or using the default
```

The static case would look like this:

```
future_t<word, static_ptr<awaitable_state<word>>> f2;
```

3.7.5.5 Outcome

There are two distinct problems with this approach.

Unfortunately, although this model did not break the existing implementation that uses `counted_ptr<>`, it did not work for the new `static_ptr<>` type. The `future_t<>::promise_type`, which is created by compiler-generated code in the coroutine ramp, contains an embedded `promise_t<>`. Clearly, this type should not be created on-the-fly for the static case, since the new instance will not point to the correct `shared_state`. (For the dynamic case, it is fine, since both the created `promise_t<>` and the original instance both use the same shared memory, which is reference counted.) As a result the program can crash when the wrong shared state is updated by the ISR. The static future **must not** create a promise during its `promise_type` construction.

This behaviour can be enabled by using `#define USE_STATIC_PTR_FOR_READ_I2C` in iteration3.

Secondly, the two forms of the future, `future_t<T, counted_ptr<awaitable_state<T>>> f1` and `future_t<T, static_ptr<awaitable_state<T>>> f2`, do not easily support combinatorial expressions, such as `f1 && f2` or `f1 || f2`, which may be required for parallel or serial invocation by composition helper classes. Indeed, any form of composition becomes significantly more complex.

3.7.5.6 Summary

This implementation of the static model works for explicitly created `promise_t<>`s but not for those created implicitly during `co_return`.

3.7.6 Fourth iteration

3.7.6.1 Objectives

- Remove remaining STL `stack<>`, used for the list of blocked coroutines.
- Remove remaining heap usage.

- Override memory allocation for `promise_type` and (via `promise_type`) for coroutine stack frame.

3.7.6.2 Remaining issues

- Remove remaining heap usage.
- Remove remaining STL dependencies.
- Override memory allocation for `promise_type` and (via `promise_type`) for coroutine stack frame.

3.7.6.3 Discussion

The fourth iteration did not succeed in all its objectives.

STL types and methods

STL types and methods remained in use as listed in Table 3.6.

Harmless. Some of these remainders are harmless:

- `std::forward()`, `std::swap()` and `std::move()` are all primitives without dangerous side-effects here. If STL were to be replaced in this library by an embedded-safe STL (similar to ETL[5]) then these methods could remain in use without changes.

Harmless in test. Some remainders would be harmful in a production build but are harmless in this test environment, and do not have a significant effect on test outcomes:

- `std::terminate()` is called in response to a caught exception, and it should be replaced by a log and reboot policy.
- `std::vector<>` is used as a container for a collection which does not change size during testing. It should be replaced by a memory-safe implementation or (if of fixed size at run-time) by a safe version of `std::array<>`.
- `std::function<>` is used as a container for a lambda function; it should be replaced by a safe version such as the ETL equivalent, `etl::function<>`.

Harmful. Some remainders are harmful and constitute a fatal design error in the library. They can only be fixed by a change to the standard:

[5] https://www.etlcpp.com/

Table 3.6: Remaining uses of STL types and methods

Type/method	Location(s)
`std::exception`	Inherited by `future_exception`
`future_exception`	Thrown by `awaitable_state<>::get_value()`
	Thrown by `promise_type<>::get_future()`
	Thrown by `spromise_type<>::get_future()`
`std::forward()`	Called by `counted_awaitable_state<>` assignment constructor
	Called by `counted_ptr<>::make_ptr()`
	Called by `counted_ptr<>::make_counted()`
	Called by `promise_t<>` assignment constructor
	Called by `static_promise_t<>` assignment constructor
	Called by `coroutine_traits<>::promise_t<>` `::return_value()`
`std::swap()`	Called by `counted_ptr<>` assignment constructor
	Called by `counted_ptr<>::operator=()`
`std::move()`	Called by `coroutine_traits<>::promise_type` `::set_exception()`
`std::terminate()`	Called by `promise_t<>::unhandled_exception()`
	Called by `spromise_t<>::unhandled_exception()`
	Called by `coroutine_traits<>::promise_type` `::unhandled_exception()`
`std::vector<>`	Contained by class `timer_item_list`
	Contained by class `event_queue_t`
`std::function<>`	Contained by class `split_phase_event_t`

- `std::exception` and classes descended from it should not be used.
- Exceptions such as `future_exception` should not be thrown.

Dynamic memory allocation

Other remaining problems concern dynamic memory allocation. Allocations are made by STL classes `std::stack<>` and `std::vector<>`. As discussed above, this use is minimal and does not significantly impact test outcomes.

Allocation by `promise_type`

The allocations made directly by `promise_type` are also small and harmless in a test environment; however, their replacement by code that avoids heap memory is a non-trivial programming task.

Finally, the potential allocation of the coroutine stack frame in heap memory is a

significant problem. Since it cannot be guaranteed to be avoided by HALO elision, there need to be changes to the standard and/or the tool-chain including:

1. a compiler flag to cause an error or warning when elision does not take place; and
2. a simple and efficient mechanism to force allocation into global (static) memory.

3.8 Appendix III: Equipment

Fig. 3.5 shows the front panel of the Rohde & Schwarz HMO2024 oscilloscope used in the study. As can be observed in Fig. 3.8, this device provided a simple, reliable and non-intrusive mechanism for measuring the performance of micro-benchmarks, without requiring the inclusion and additinal complexity of software timing code.

Fig. 3.6 shows a Freescale FRDM-K22F development board, as used in the study. This board, first produced in 2014, was used as the main board for teaching the JCU undergraduate courses in Embedded Systems until 2020: its hardware and development tools were therefore familiar to the authors.

Fig. 3.7 contains a diagram of the simple circuitry used to connect the oscilloscope to the development board.

Figure 3.5: The front-panel display of the Rohde & Schwarz HMO2024 oscilloscope.

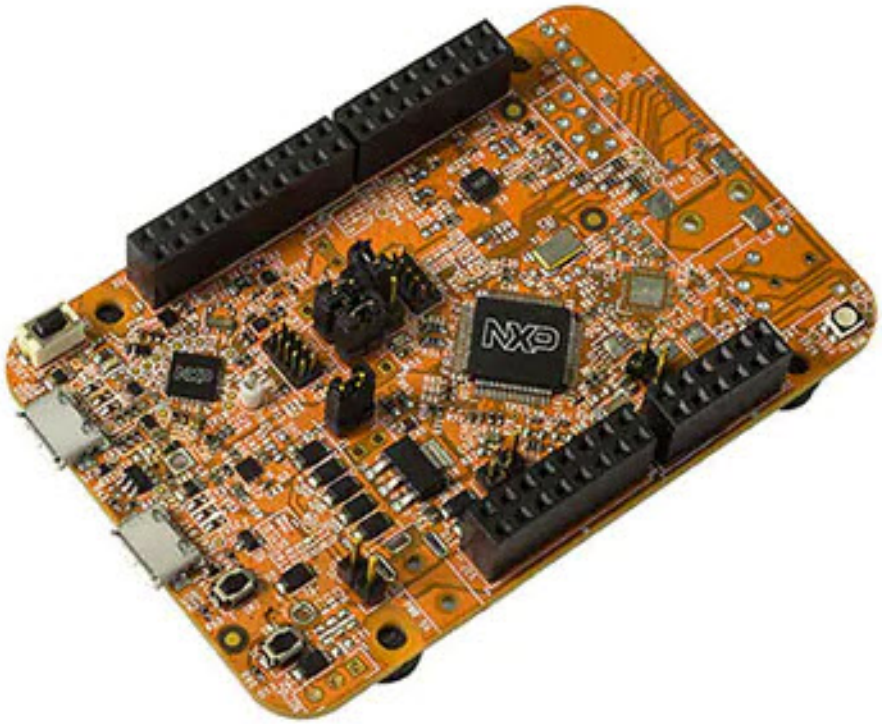


Figure 3.6: A Freescale FRDM-K22F development board, as used in the study.

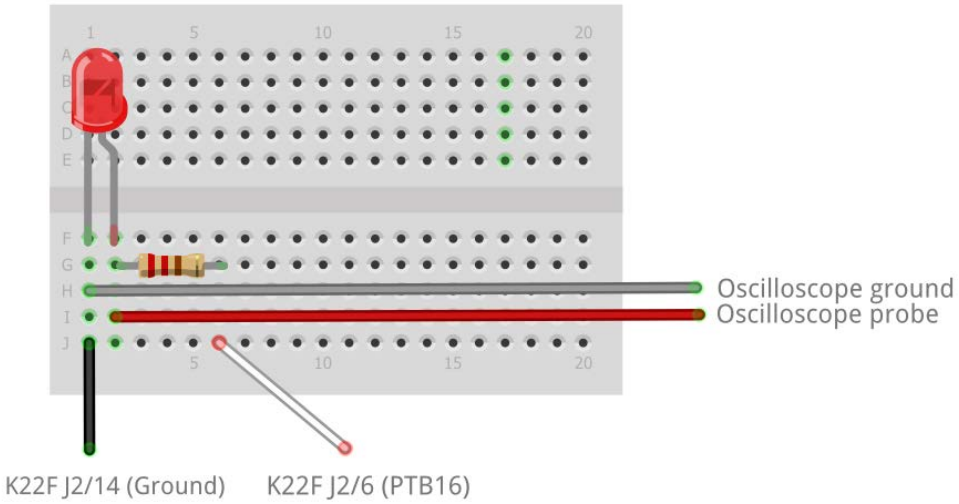


Figure 3.7: A diagram of the simple breadboarded circuit used to connect the oscilloscope to the development board.

3.9 Appendix IV: Data

Typical screen captures recorded by the Rohde & Schwarz HMO2024 oscilloscope are shown in Fig. 3.8.

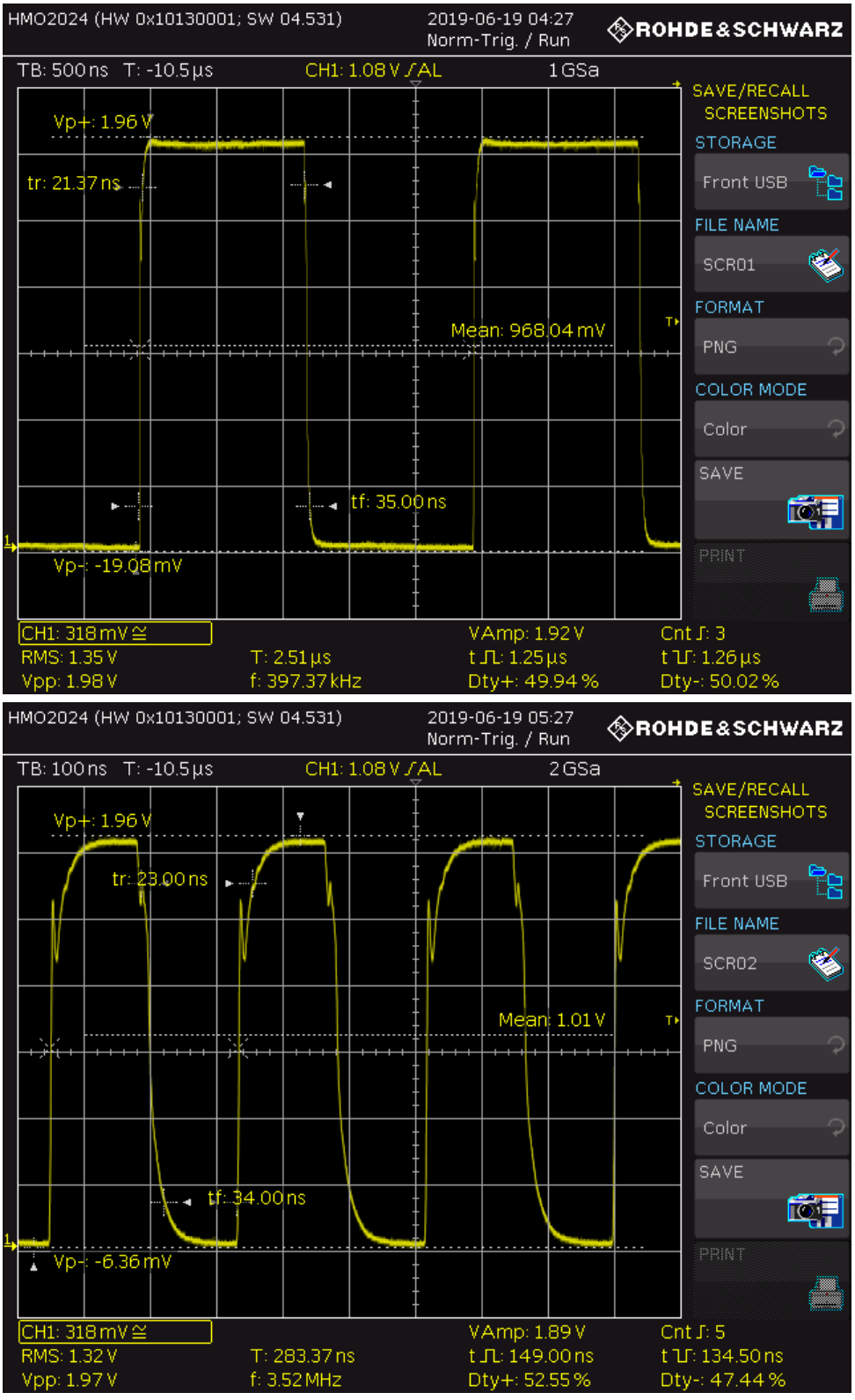


Figure 3.8: Typical screen captures recorded by the Rohde & Schwarz HMO2024 oscilloscope. Each graph shows the voltage detected between the active and ground GPIO pins of the FRDM-K22F development board. The screen captures illustrate the oscilloscope's built-in capability for measuring the frequency of square waves.

Chapter 4

Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines

This chapter is an expanded and reformatted version of the following paper for the IEEE Conference *25th IEEE International Conference on Computational Science and Engineering (CSE 2022)*.

> B. Belson and B. Philippa, "Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines", in *Proceedings of the 2022 IEEE 25th International Conference on Computational Science and Engineering (CSE)*, doi: 10.1109/CSE57773.2022.00011.

The paper was published in November 2022, in a conference format limited to eight pages. This chapter therefore contains significant amounts of extra material which were excluded from the conference paper, including figures presented at the conference, more detailed results and some discussions of implementation issues.

Chapter Abstract

We demonstrate a novel method of speeding up large iterative tasks such as machine learning inference. Our approach is to improve the memory access pattern, taking advantage of coroutines as a programming language feature to minimise the developer effort and reduce code complexity. We evaluate our approach using a comprehensive set of benchmarks run on three hardware platforms (one ARM and two Intel CPUs). The best observed performance boosts were 65% for scanning the nodes in a B+ tree, 34% for support vector machine inference, 12% for image pixel normalisation, and 15.5% for two dimensional convolution. Performance varied with data size, numeric type, and other factors, but overall the method is practical and can lead to significant improvements for edge computing.

4.1 Introduction

The emergence of the Internet of Things and the associated increase in the volume of data available to devices at the edge of the Internet have created a demand for machine learning (ML) inference on relatively low-powered, resource-constrained devices [160, 200]. ML inference on local edge devices has received increasing attention due to the benefits of reducing network transmission delays, reducing the required network bandwidth, potentially increasing privacy, and improving resilience against network outages [29]. However, running ML inference on the edge can be more challenging than in the cloud due to the more limited computational resources that are likely to be available. Therefore, considerable research attention has been devoted towards machine learning on the edge [128].

Various strategies exist to improve the performance of ML inference, including the development of specialised accelerator hardware [86, 199, 32, 31, 61], tool-chain enhancements [28, 199, 191, 109], architectural improvements [26] and hybrid approaches [22, 199]. The performance of ML algorithms can also be improved by re-ordering the code in such a way as to benefit from CPU cache locality and memory prefetching [44, 12]. We refer to this as the memory access pattern of a program.

Improvements to the memory access pattern can lead to large performance increases, and is widely used in specialised code that is hand-written for performance [30, 75, 146].

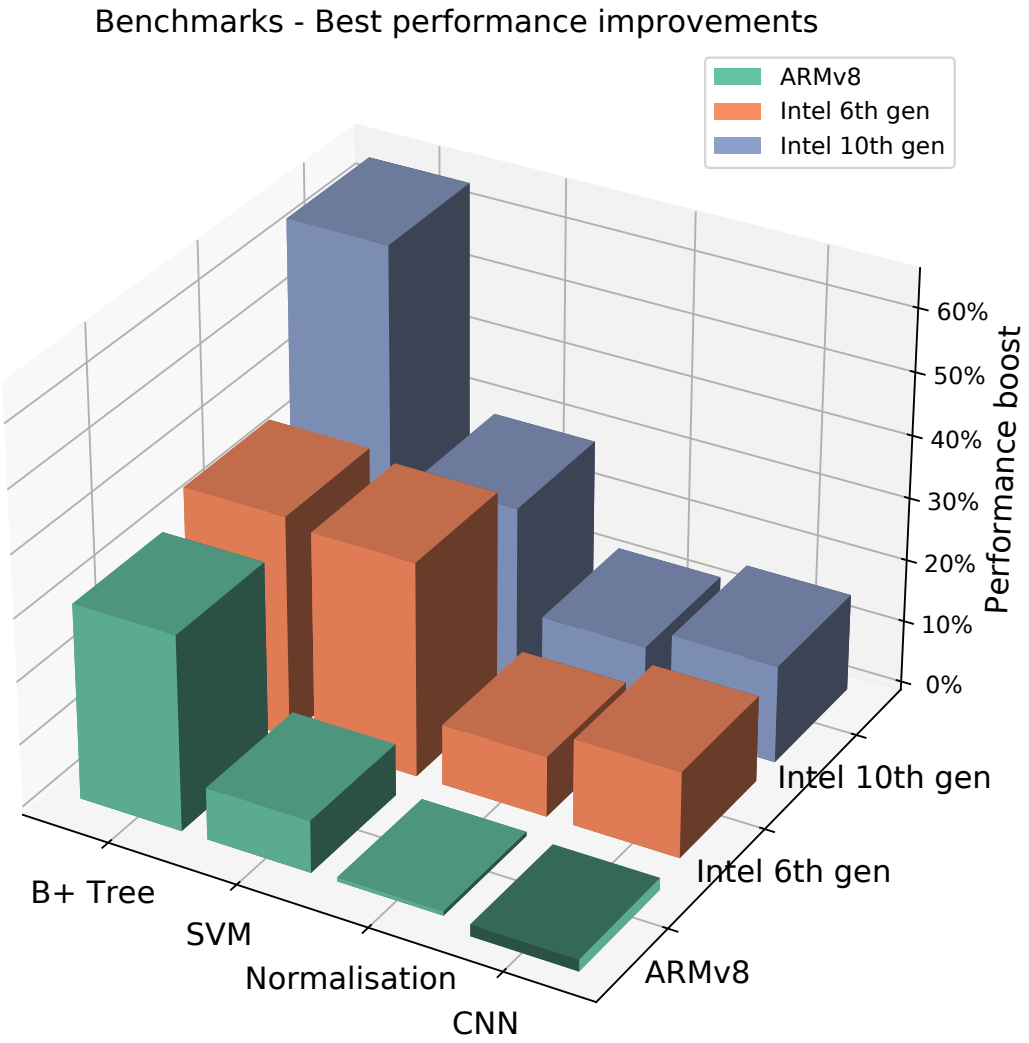


Figure 4.1: Best performance improvements achieved through coroutining and prefetching, across various algorithms and platforms

However, such an approach can be difficult to retro-fit into existing code without a major rewrite.

In this work, we show how to improve the memory access pattern of software without dramatic changes to its source code by leveraging programming language features for asynchronous programming. Specifically, we utilise coroutines as a way to more easily re-order an existing algorithm to improve its caching and memory prefetching performance.

A coroutine is a subroutine that can be suspended (by saving its execution state) and resumed at a later time [17]. Coroutines are present in C++, C#, Go, Python, Rust, and many other languages. In this work, we demonstrate how straightforward implementations of common algorithms can be optimised, without compromising the clarity of the code, by leveraging the compiler's coroutine support to reorder the flow of execu-

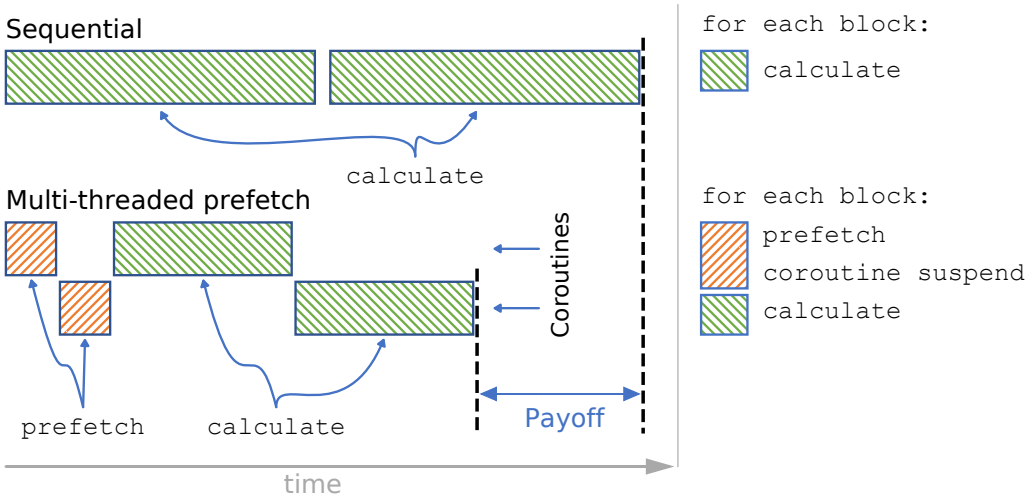


Figure 4.2: Execution models compared. In the standard sequential execution model, cache misses freeze the thread until the target memory region is loaded into cache. Using multi-threaded prefetch, the target memory region is requested and the thread is then suspended; when it resumes the memory is available in cache. If the benefit of avoiding cache misses is greater than the cost of the coroutine infrastructure, then the transformation has a positive payoff.

tion. An overview of our approach is shown in Fig. 4.2. A sequential algorithm can be transformed into one with an interleaved memory pattern by using the coroutine task switching functionality in modern programming languages. The result is a small change to the code structure, since the complexity in handling the interleaved state is managed by the compiler.

The contributions of our work are as follows:

1. We demonstrate a novel approach to optimising algorithms that are common in edge computing, achieving in the best case up to 65% speed up. These results are summarised in Fig. 4.1

2. We conduct a comprehensive set of benchmarks to assess the performance of this approach, on four algorithms, three hardware platforms, two compilers, and four different numeric data types. Our benchmarks also assess the costs of different compiler implementations of C++20 coroutines compared to alternative lightweight thread approaches.

3. We examine the development effort and challenges involved in applying the transformation to an existing code base.

The remainder of this chapter is organised as follows. Section 4.2 contains the back-

ground and related work. Section 4.3 details the experimental methodology. Section 4.4 explores the results and section 4.5 discusses and classifies the outcomes. Section 4.6 concludes the chapter.

4.2 Related work

As memory models have become increasingly complicated and more dynamic, it has become harder for applications programmers to take maximum advantage of the benefits that locality might bring [12, 22]. The performance of a data-intensive algorithm can be extremely sensitive to data cache misses. For example, a counterintuitive rearrangement of the loops in a standard naive algorithm for matrix multiplication [169] can reduce data cache misses significantly, and improve performance by 10-20%.

As CPU caches have become more central to performance, model-dependent software prefetching techniques have been established and refined in specific application areas such as matrix LU decomposition [127], pointer chasing [30] and across general application fields [188]. However, such software is generally designed from the ground up to consider the memory access pattern, and execute prefetch instructions at the right time. The programmer is typically responsible for this book-keeping, which increases the effort required in implementation and testing.

The key advantage of using coroutines is that the compiler helps reorder the flow of execution. Previous work has demonstrated the use of coroutines to schedule memory prefetching on large server platforms in the cloud, for example, with in-memory database engines [145, 85, 75]. Coroutine based prefetching has also been used to speed up software-defined networking [8]. Language support has been recognised as important, for example, Kiriansky *et al* introduced a domain-specific language to more easily express the required source code transformations [92]. It is notable that all of these applications targeted heavily multi-core server platforms. The benefits for lower-powered platforms have not yet been widely studied, despite the fact that coroutines are very lightweight and are suitable even for embedded microcontrollers [18]. To the best of our knowledge, this work is the first to demonstrate that coroutine based prefetching can be applied to resource-constrained edge devices.

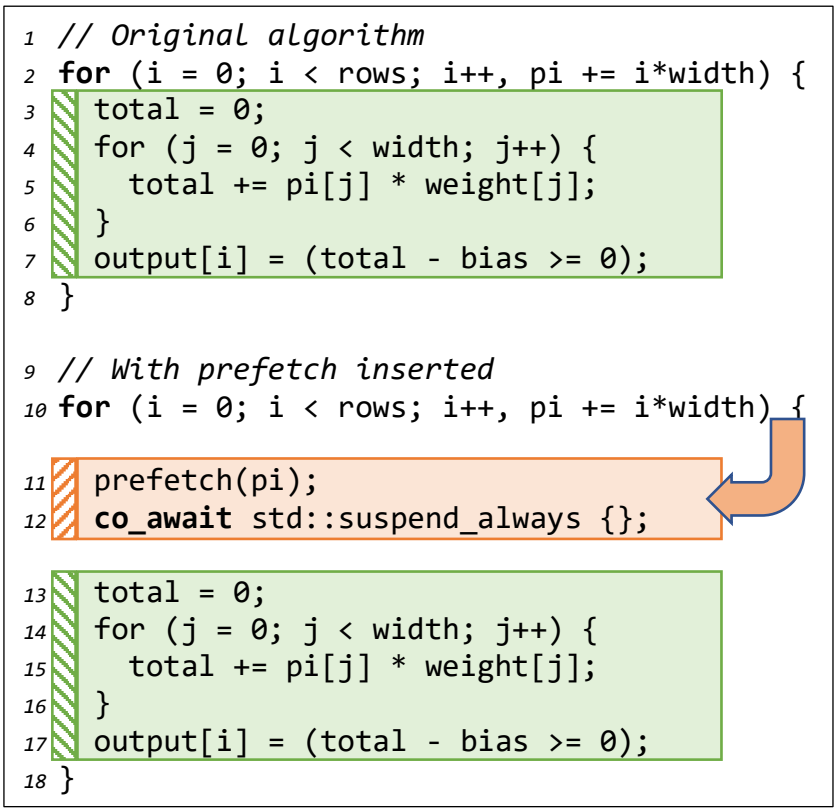


Figure 4.3: Support vector machine inference code in C++, showing code versions before and after the prefetch code is added.

4.3 Methodology

4.3.1 Implementation

An overview of our implementation is shown in Fig. 4.3. The figure shows C++ code for a support vector machine (SVM). The original algorithm is a simple loop over a table of observations, where each observation needs to be processed through the SVM. The modified code shows how straightforward it is to add coroutine-based prefetching. In the modification, line 11 initiates a prefetch for the next chunk of data and line 12 suspends execution. In this way, the coroutine cooperatively yields to allow another data buffer to be processed. By the time the task is resumed (line 13), the next data required for this iteration will be in the CPU memory cache.

This approach was designed to be as simple as possible to retrofit to existing code or, alternatively, as simple as possible for the programmer to use in new code. The algorithm can be written directly, and all the book-keeping to handle the data interleaving is generated automatically at compile time. Therefore, this approach is facilitated by coroutines as a first-class language feature. If such support is present in the language,

Table 4.1: Benchmarks used for performance measurement

Name	Description	Output
B+Tree	Visit the leaves of a B+ tree	Reduce[1]
SVM	Support vector machine inference	Reduce[1]
Norm	Normalise image to ImageNet mean	Map[2]
CNN	Two-dimensional convolution	Map[2]

[1] Output dimensionality is reduced from input

[2] Output dimensionality is the same as input

then this type of prefetching is highly practical for developers to use.

4.3.2 Benchmarks

We measured the performance of our approach by applying it to four distinct microbenchmarks, as listed in Table 4.1. These benchmarks are as follows.

B+ Tree visits each leaf node, scans through the data stored in that node, then follows a pointer to the next node, and repeats. The nodes are laid out pseudo-randomly, as might be expected if the B+ tree grew organically with data being inserted in an arbitrary order.

Support Vector Machine (*SVM*) examines a set of vectors and calculates a weighted sum for each, then compares it to a bias value.

Normalisation (*Norm*) rescales an image by subtracting the mean pixel value of ImageNet [43] and dividing by the standard deviation.

Convolutional Neural Network (*CNN*) applies a 3x3 kernel to the pixels of a batch of monochrome images.

For the benchmarks that used numbers (*SVM*, *Norm* and *CNN*), we tested 16 bit integers (which we denote *i16*), 32-bit integers (*i32*), single precision floats (*f32*), and double precision floats (*f64*). For the integers, we used fixed point format with scaling factors of 2^{13} and 2^{26} for *i16* and *i32*, respectively. The numeric types are summarised in Table 4.2.

4.3.3 Performance measurement

For each test, we measured a standard (sequential) implementation and a modified version of the same code that uses coroutines to reorganise the memory access. We

Table 4.2: Numeric types

Name	Type	Bytes	Format	C type
i16	Fixed-point	2	Scaling factor 2^{13}	short int
i32	Fixed-point	4	Scaling factor 2^{26}	int
f32	Floating-point	4	IEEE 754	float
f64	Floating-point	8	IEEE 754	double

Table 4.3: Active set variables

Benchmark	Factors varied
B+Tree	Tree size, branching factor
SVM	Columns per data table, number of data tables
Norm	Rows per image, columns per image, number of images
CNN	Rows per image, columns per image, number of images

define the *performance ratio* as

$$\text{Performance ratio} = \frac{Z_{sequential}}{Z_{modified}}$$

where Z is the number of CPU cycles used by the process. A ratio $> 100\%$ corresponds to an improvement in performance.

For an implementation with M coroutines, each using N bytes of input and output data, we define the *active set* size to be $M \times N$. We will show below that the active set size needs to be similar to the cache size in order to maximise the benefit of our approach.

For each algorithm, on each hardware platform, we explored a large factor space. We varied the number of concurrent tasks; we used several execution models, including the standard sequential model, a coroutine model, a coroutine model with prefetch and a Protothreads[49] model, with and without prefetch; and we varied the size of the active set by controlling the size of each level of iteration for each task. Protothreads provides lightweight cooperative multitasking in C; it is stackless, does not maintain the state of local variables, and uses non-standard C. It was included here in order to test the benefits of light-weight task-switching without the same machinery cost as language-native coroutines. This offered a direct comparison to estimate the cost of the coroutine machinery as distinct from the task switching. We also tested the impact of including explicit prefetch instructions, or simply interleaving the data processing without issuing

prefetch commands.

The size of the active set was controlled for each algorithm as follows. For the B+ tree, we varied both the tree size and the branching factor. For the normalisation and CNN tests, we controlled the number of rows and columns in each image and the number of concurrently processed images. For SVM, we varied the number of columns in the data table, and the number of data tables. The variables are listed in 4.3.

We repeated all measurements 10 times. A proportion ($\approx$ 18%) of measurements appeared to be outliers. We assume they are caused by competition for cache resources from background tasks running on the same computer. Therefore, we removed any measurement whose variation from the median was $> 3.5\sigma$, where σ was the mean variation from the median.

4.3.3.1 Telemetry

The Linux *perf_event_open()* API was used as the basis for telemetry. We measured the hardware counters *cpu-cycles*, *instructions*, *cache-misses* and *cache-references*. This approach isolated the performance of the process under test, where a simple timer would always be impacted by the various other tasks and daemons of the operating system. We avoided using platform-specific counters in order to provide consistency across the different hardware platforms under test.

4.3.4 Platforms and Toolchains

Our experiments tested three platforms that are representative of the edge computing devices where significant ML inference would be expected to take place. These platforms were the Raspberry Pi 4 (containing an ARM Cortex A72 microprocessor and released in 2020), as well as commodity PC hardware containing a 6th generation Intel i7-6700k (2015) and a 10th generation Intel i5-10500T (2020). The cache sizes for each platform are shown in Table 4.4.

We used Ubuntu 18.04 & 20.04 respectively on the Intel 6^{th} & 10^{th} generation machines, and Raspbian GNU/Linux 10 (buster) on the ARM CPU. We tested two compilers: GCC 10.2 and LLVM 11.0.

Table 4.4: Platforms & memory caches tested

CPU	L1	L2	L3	Total
ARM Cortex-A72	32 KiB	1 MiB	-	1056 KiB
Intel i7-6700k	128 KiB	1 MiB	8 MiB	9344 KiB
Intel i5-10500T	192 KiB	1.5 MiB	12 MiB	14016 KiB

Table 4.5: Benchmarks - Best results for each algorithm

	B+ Tree	**SVM**	**Norm**	**CNN**
ARMv8	✓31.1%	✓8.3%	✓0.7%	-1.9%
Intel 6^{th}-gen	✓35.5%	✓34.2%	✓9.6%	✓13.7%
Intel 10^{th}-gen	✓64.8%	✓28.8%	✓12.4%	✓15.5%
Factors	Prefetch Randomness Branching factor	Prefetch Numeric type	Numeric type	Numeric type Image width

The number of bytes processed per CPU cycle was compared between the standard and coroutined execution models. The difference in performance is shown as a percentage, with a positive result indicating a boost in performance using a coroutined execution model.

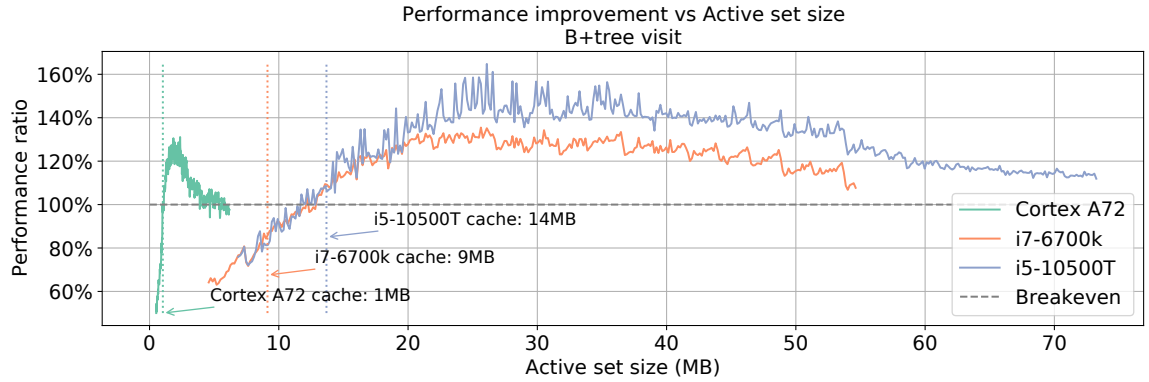
4.4 Experimental results

Table 4.5 shows the highest performance ratios observed for each of the benchmarks tested, across all numeric types, number of concurrent coroutines, active set sizes and with or without explicit prefetch instructions. It can be seen that in almost all cases, there exists a point in the parameter space where the coroutine-based implementation was faster than the control case (the unmodified sequential algorithm). These results are also summarised in Fig. 4.1.

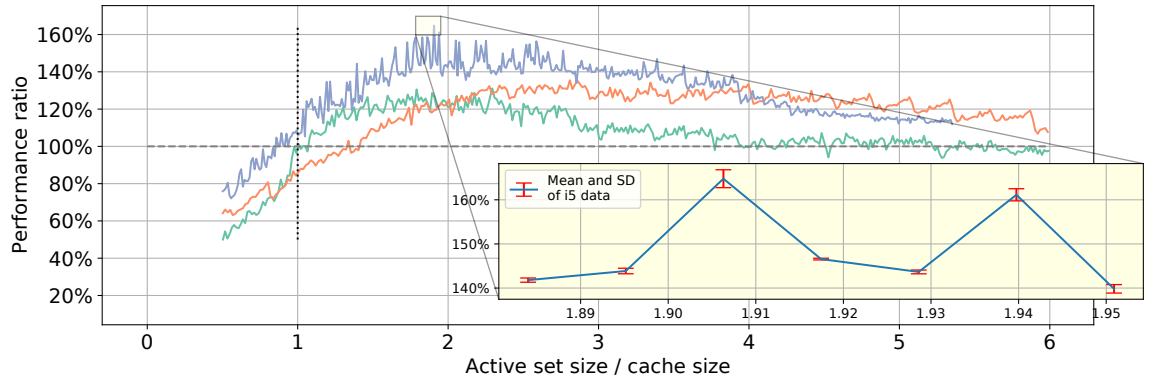
4.4.1 Impact of active set size

The parameter that has the largest impact is the size of the active set. Recall that the active set is the total amount of memory being used across all coroutine tasks, and can be tuned by changing the number of coroutines and the amount of memory processed by each coroutine during a single iteration of the resume-calculate-suspend loop.

Fig. 4.4 shows the typical response to variations in the active set size. All platforms display similar behaviour: for small active set sizes the technique imposes a high cost;



(a) Raw data for each platform



(b) Normalised by total cache size

Figure 4.4: Performance ratios for varying active set sizes. **(a)** Raw data for each platform. **(b)** The same data but with the horizontal axis normalised by total cache size. The inset shows a zoomed in region near the peak of the curve with error bars representing the standard deviation of repeated measurements.

as set size approaches total cache size, we begin to see improved performance, rising to a maximum at 2x to 3x cache size; performance gains then fall as set size rises further, reflecting a flooded cache.

There is thus a 'sweet spot' for this algorithm between 1x and 3x-6x the cache size (depending on platform) where very large performance gains are achievable. For a developer this behaviour offers a simple and powerful optimisation technique: once the coding changes have been implemented, the size of the mini-batch can be manipulated so that the algorithm operates within the 'sweet spot'. Potentially, such optimisation could be performed automatically for a given machine learning inference algorithm.

4.4.2 Sensitivity to active set size

Fig. 4.4 contains a rather 'spiky' data series. These spikes are not due to experimental error: the variation between repetitions is smaller than the size of the 'spikes' (as per

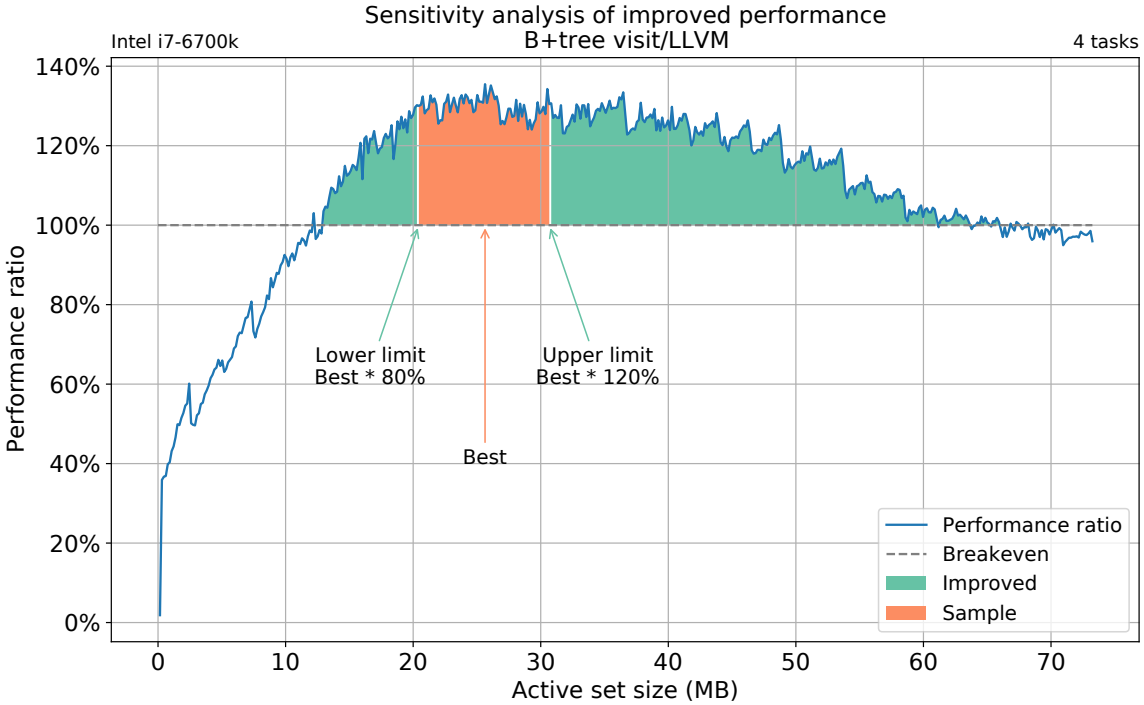


Figure 4.5: Sensitivity estimation: each test point is run 10 times, and outliers $> 3.5\sigma$ (as per the Methodology section) are removed; the data size for the best performance is identified; the region which uses $\pm 20\%$ of this size is extracted and analysed.

the inset of Fig. 4.4b). While the programmer has some control of the size of the active set, some factors such as the width of an image may not be adjustable. Therefore, the precisely optimal active set size may not always be achievable.

We examined the sensitivity to the data size by investigating how much performance varied as the active set size was varied from the optimal value by $\pm 20\%$. This window is shown visually in Fig. 4.5. The sample of performance ratios observed within this $\pm 20\%$ window represent the range of variation that might occur if the optimal active set size is not achieved exactly, and hence shows the robustness of the optimisation technique.

The impact of variations in the active set size is shown in Fig. 4.6, Fig. 4.7, Fig. 4.8 and Fig. 4.9. The figures show all four algorithms, all four numeric data types (except for B+ Tree), and all three hardware platforms. In the plot, positions further to the right indicate a performance boost. Furthermore, the larger the range shown, the greater is the likelihood that the performance gains will be unpredictable (if the active set size is not precisely known).

The box-and-whisker plots show statistics about the region highlighted in Fig 4.5. A long bar indicates a highly variable performance gain: if the bar crosses the 100% mark, then a developer cannot be confident of a benefit from using these techniques unless

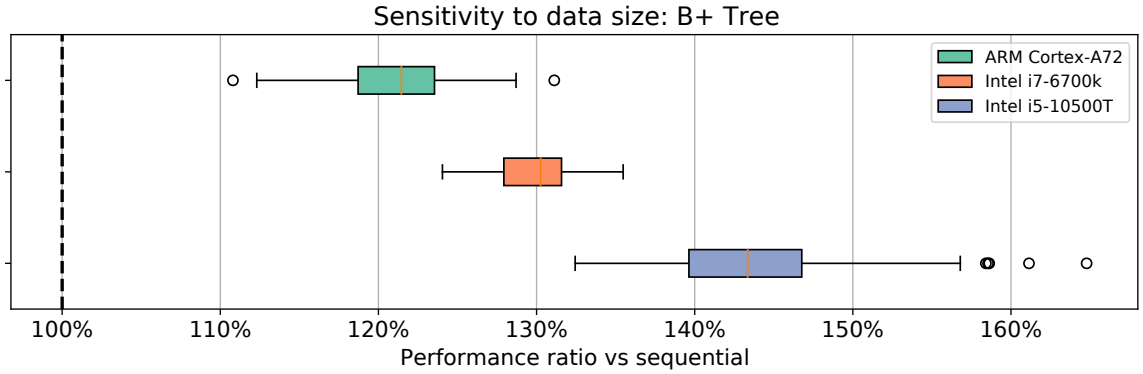


Figure 4.6: B+ Tree: Sensitivity of performance gain to data size. Substantial performance gains are visible on each platform, and also a high degree of confidence that the gain will be within a well-defined band.

the size of the active set can be precisely controlled. Notice the varying horizontal axis scales.

For the B+ Tree, Fig. 4.6 shows a substantial performance gain on each platform, but also implies a high degree of confidence that the gain will be within a well-defined band.

For SVM, Fig. 4.7 indicates a performance gain that is strongly influenced by numeric type. A 16-bit fixed-point numeric type provides a solid performance gain for all three platforms. For the 32-bit fixed-point numeric type, performance gains appear across most quartiles, but not all. Floating point numeric types consistently display losses in performance.

For Normalisation, Fig. 4.8 shows performance gains only for Intel platforms and only for floating point numeric types.

Finally, for the CNN algorithm, Fig. 4.9 shows performance gains only for the Intel platforms and only for 16-bit fixed-point numeric types.

4.4.3 Impact of coroutine count

The number of parallel coroutines was varied between 2 and 8. The impact of the coroutine count was minimal; in most cases, if an algorithm was boosted by coroutine use, the highest boost was achieved with 2 coroutines.

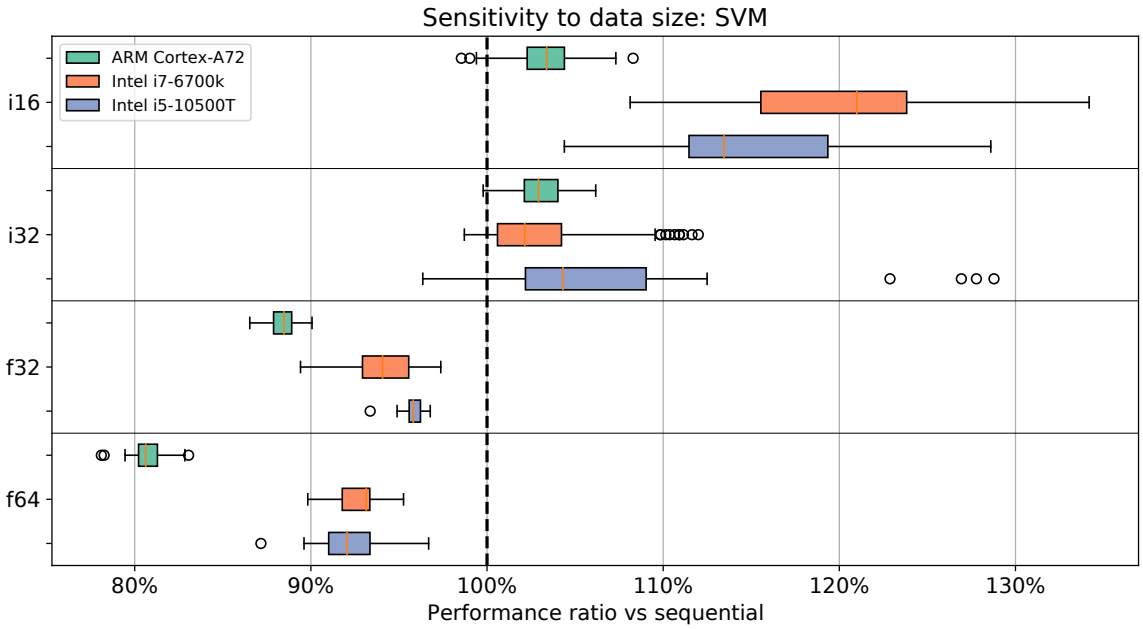


Figure 4.7: The SVM algorithm's sensitivity of performance gain to data size is strongly influenced by numeric type. A 16-bit fixed-point numeric type provides a solid performance gain for all three platforms. For the 32-bit fixed-point numeric type, performance gains appear across most quartiles, but not all. Floating point numeric types consistently display losses in performance.

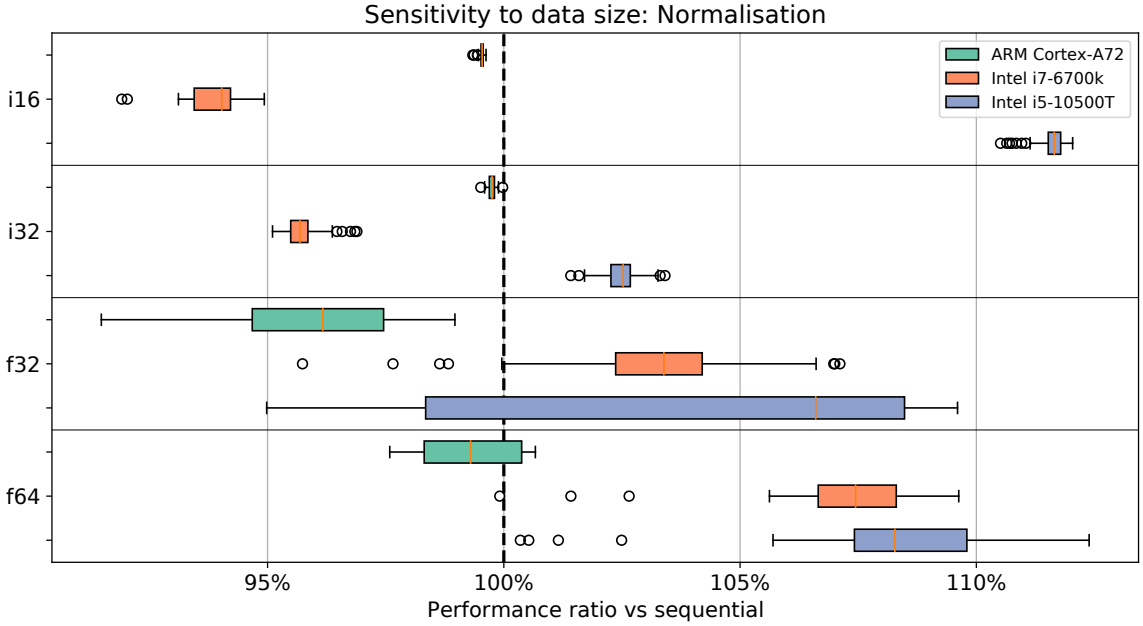


Figure 4.8: Normalisation: Sensitivity of performance gain to data size. Performance gains are seen only for Intel platforms and only for floating point numeric types.

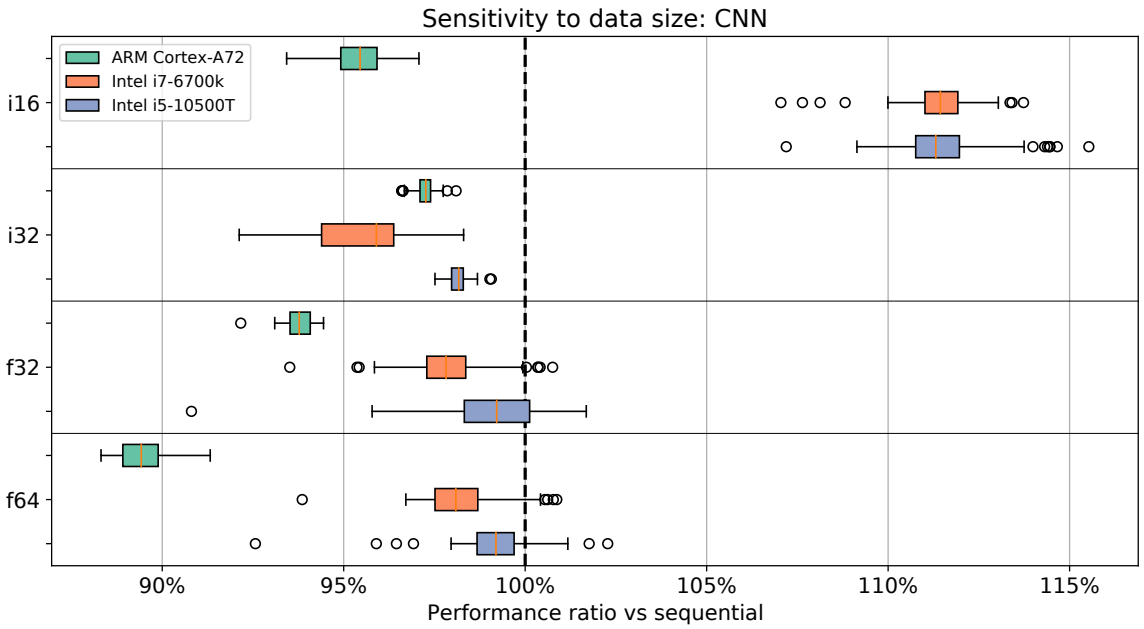


Figure 4.9: CNN: Sensitivity of performance gain to data size. Performance gains are found only for the Intel platforms and only for 16-bit fixed-point numeric types.

4.4.4 Impact of numeric types

The impact of numeric types varied across algorithms: this is illustrated by Figs. 4.7, 4.8 and 4.9, where better performance gains are displayed as bars further to the right. In each chart the outcomes for a specific numeric type across the three platforms are grouped together. In all cases where a measurable boost occurred, fixed-point arithmetic using 16-bit values offered the same or better performance boosts than 32-bit. The techniques did not offer a benefit for the floating-point versions of SVM or CNN, but did provide a benefit for floating-point normalisation.

4.4.5 Variations between algorithms

4.4.5.1 B+ Tree

The B+ tree algorithm was notable both for the simplicity of the code transformation required to introduce asynchronous prefetching and for the clear performance benefits achieved, as shown in Fig. 4.10.

Note that the performance benefits disappear if the B+ tree was carefully constructed in such a way that the nodes are laid on (on the heap) in a sequential order, such that the end of one leaf is near to the beginning of the next. In this situation, the control case (the sequential algorithm) runs much faster, and there is no opportunity to further

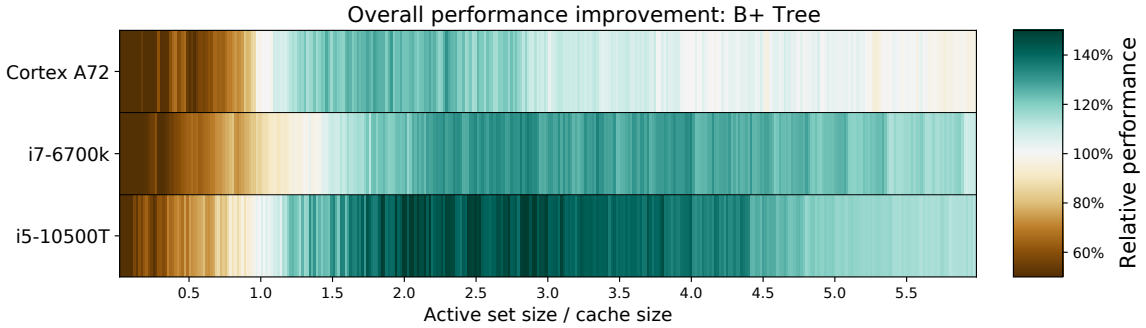


Figure 4.10: Performance boost achieved for B+ Tree algorithm. Compared to the standard sequential execution model, a model using coroutine with prefetch performed up to 50% faster.

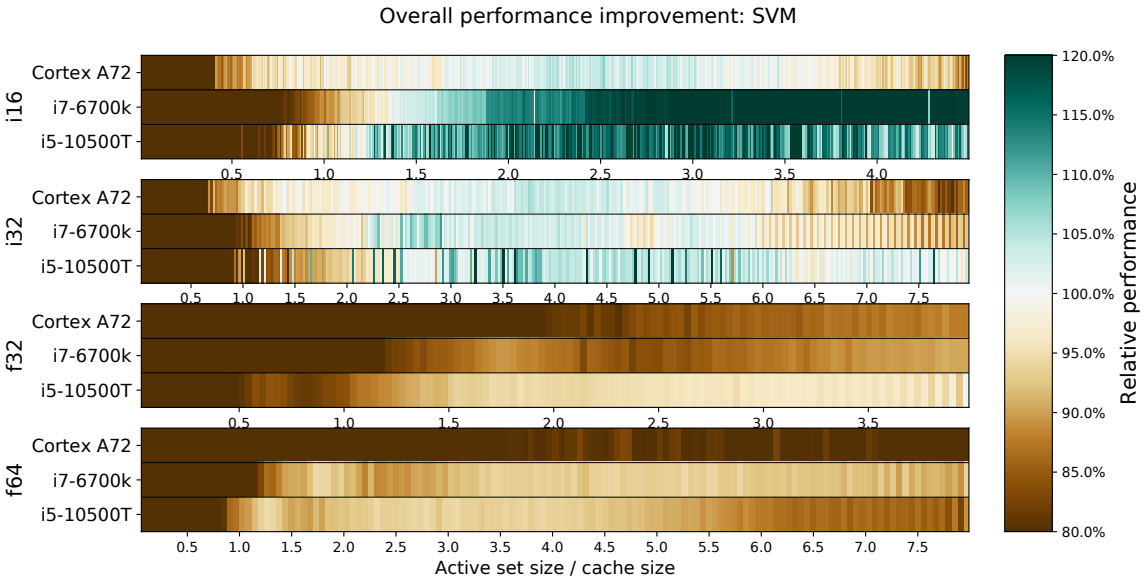


Figure 4.11: Performance boost achieved for SVM. All platforms show gains with fixed-point numbers, while floating-point numbers offer benefits on none.

optimise the memory access pattern. We attribute this to the CPU cache being optimised for sequential reads of large blocks of memory [170].

Our approach adds benefit when the heap layout of the B+ tree is non-sequential. These findings reflect the work of Chai *et al.* [28] whose proposed cost model calculates prefetch revenue according to the distance of memory jumps.

4.4.5.2 Support Vector Machine

The SVM algorithm displayed useful performance boosts for all platforms when using integer (fixed-point) arithmetic, as shown in Fig. 4.11. The addition of prefetch added an additional 8-18% boost, making this an exemplary use case for both modifications, as discussed in Section 4.4.7. When floating-point arithmetic was applied to the model the

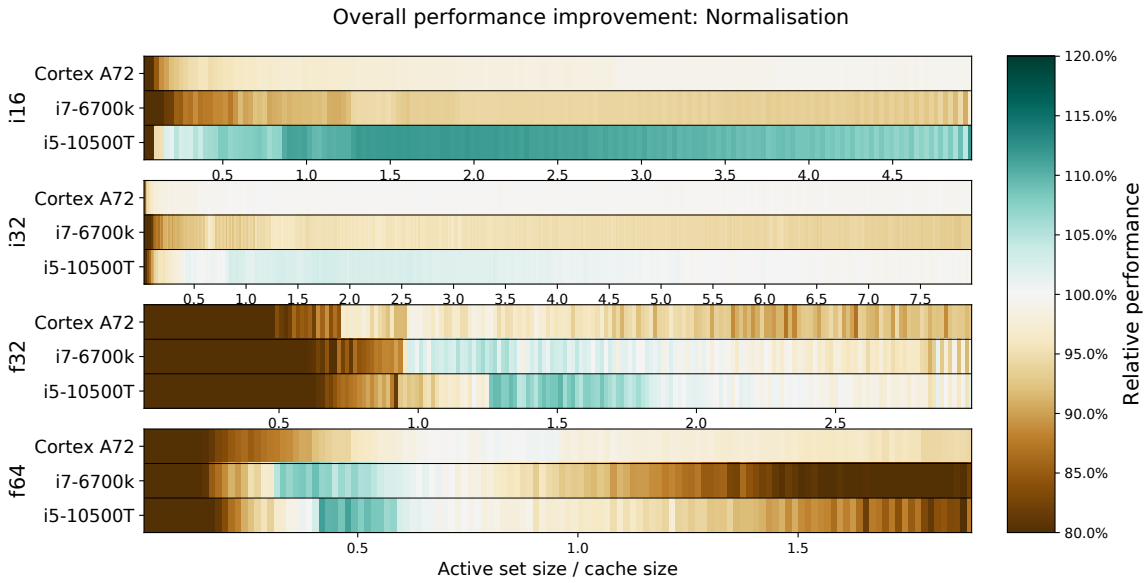


Figure 4.12: Performance boost achieved for colour normalisation algorithm. Benefits varied widely across platform and numeric type. The 10th generation Intel CPU boosts significantly better than the 6th generation CPU.

benefit from improved memory access patterns was slightly reduced, and the benefit from prefetching was zero or negative.

4.4.5.3 Normalisation

The normalisation algorithm is capable of providing performance improvements on both Intel platforms across a varying range of data sizes, as shown in Fig. 4.12.

The benefit of prefetching was near zero for this algorithm when fixed-point numbers were used: any performance boosts were a result of improved memory access patterns. However, there was evidence of an appreciable performance boost (9% - 12%) contributed by prefetching for the floating-point versions on the Intel architectures.

4.4.5.4 Convolutional Neural Network

The CNN algorithm showed significant performance benefits of between 12% and 13.5% on the Intel platforms when used with 16-bit fixed-point numbers and the LLVM compiler, as shown in Fig. 4.13. No other combination showed any benefits.

The benefit of explicit prefetching was near zero for this algorithm: the performance boost was a result of changing memory access patterns through coroutining.

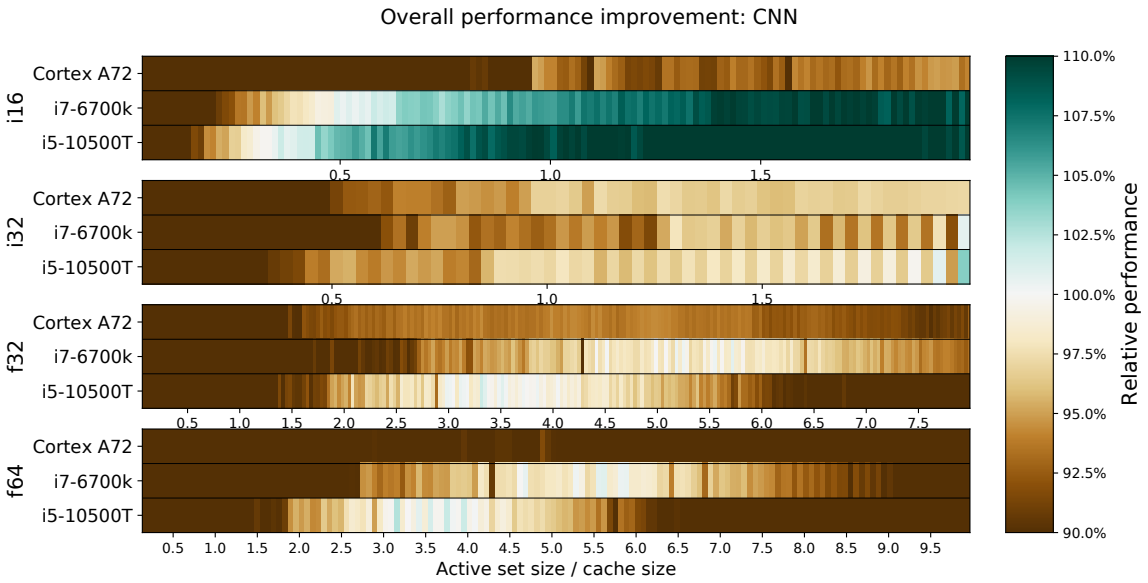


Figure 4.13: Performance boost achieved for convolutional neural network. The boost was seen only on Intel platforms and only with 16-bit fixed-point number types.

4.4.6 Platforms

4.4.6.1 Intel platforms

Both Intel platforms showed performance boosts across all four algorithms; on the newer 10^{th} generation CPU these boosts were particularly significant: between 12.4% and 64.8%.

4.4.6.2 ARM platform

The ARM platform showed very significant boosts (up to 31%) for the pointer-chasing algorithm, *B+ Tree*. The numeric reducing algorithm *SVM* showed an appreciable improvement (8%) using coroutining with prefetching.

The numeric mapping process *Norm* also showed some benefits (up to 0.7%); prefetching was not required - the boost was achieved entirely through improved memory access patterns. The numerically intensive convolutional neural network benchmark (*CNN*) was not significantly improved by any combination of execution model or numeric type.

4.4.7 Impact of explicit prefetch instructions

Prefetching offered significant improvements when applied to the algorithms which reduced the data dimensionality - *B+-Tree* and *SVM*. Prefetch provided all of the performance benefits observed in *B+-Tree* and boosted *SVM* performance by a further 8-18%.

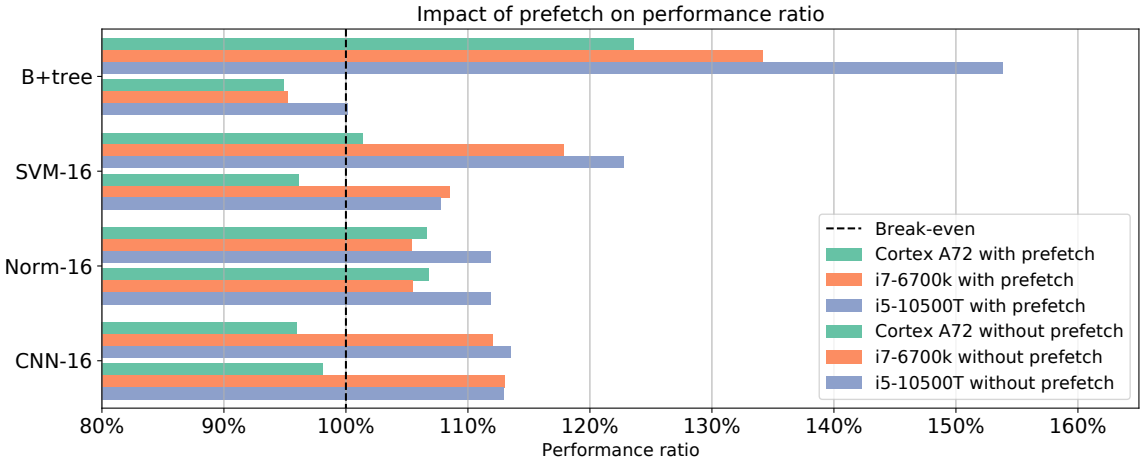


Figure 4.14: Performance boost achieved for each algorithm using 16-bit fixed-point arithmetic. Performance gains varied substantially according to algorithm and platform. For the Reduce operations (where output dimensionality is lower than input), prefetch improved performance.

However, prefetching generally offered little benefit for mapping operations (*Norm* and *CNN*), where the output's dimensionality was the same as the input's: any performance boost was a result of improved memory access patterns. (An exception was observed for floating-point operations with *Norm*, where prefetching offered performance benefits of $\approx 5\%$.)

Fig. 4.14 shows more details of the performance boost offered by the two techniques. It illustrates the difference between performance benefits resulting purely from splitting a process into multiple parallel units of execution, and those achieved through prefetching.

4.4.8 Toolchain

The LLVM toolchain consistently showed similar or better performance boosts than the GCC toolchain. Fig. 4.16 compares the peak performance boosts for each algorithm on each platform.

It is illuminating to compare the actual performance of the compilers with regard to raw data throughput, rather than the performance improvement achieved by applying the coroutine algorithms. Fig. 4.15 shows the data throughput of all execution models under each compiler, applied to the B+ Tree test. (Note that these throughputs are expressed in bytes per CPU cycle, rather than bytes per second. This approach removes the impact of any other tasks that may be running on these multi-tasking platforms and

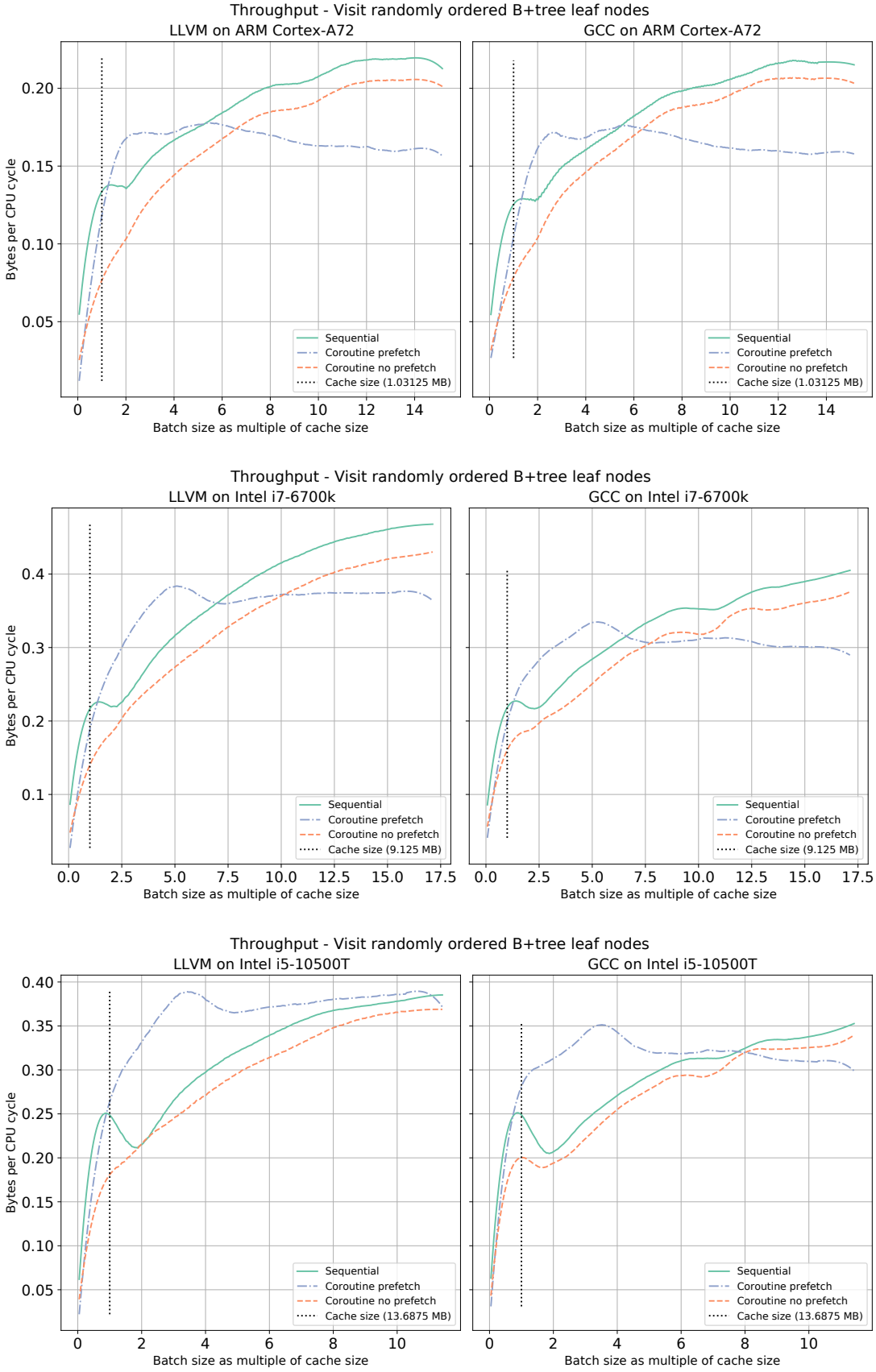


Figure 4.15: Comparison of data throughput for each compiler. Each figure shows the throughput (expressed as bytes per CPU cycle) vs the batch size (expressed as the a multiple of cache size on the platform) for a specific platform.

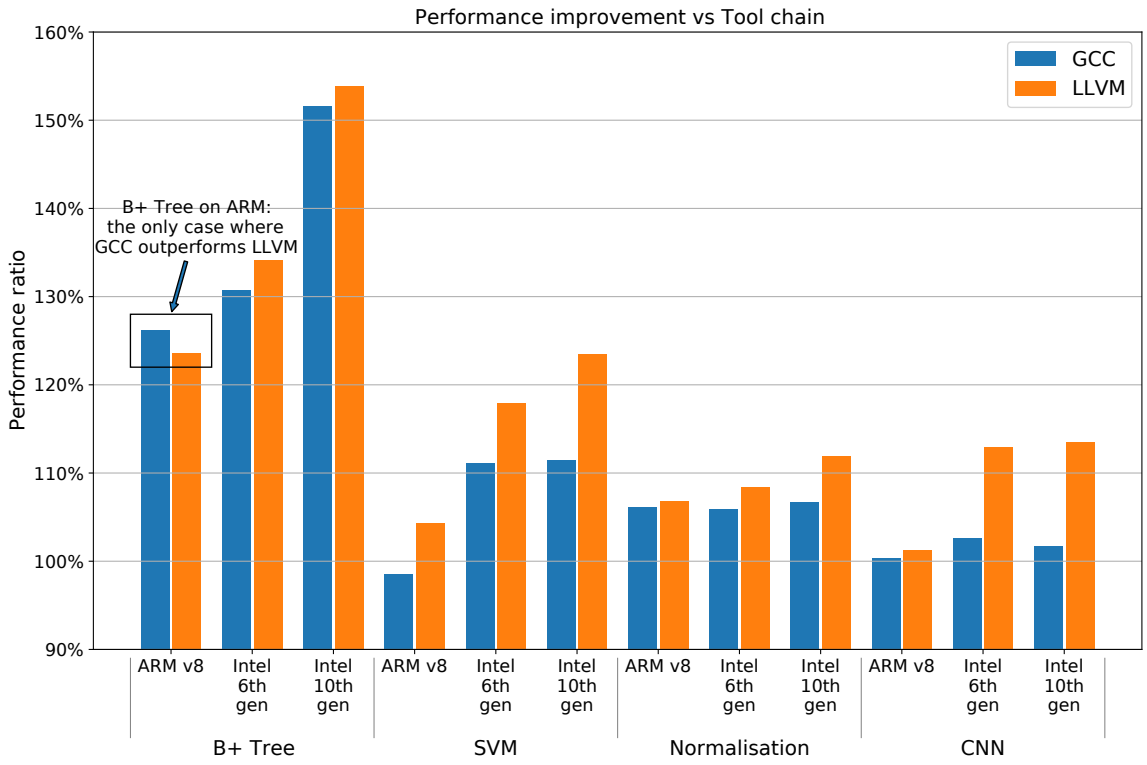


Figure 4.16: Performance boost achieved for each algorithm using each toolchain. Observe that the GCC toolchain offered superior performance gain over LLVM in only one case - B+ Tree on the ARM processor.

normalises the performance across the three different platforms.)

We observe that on the ARM Cortex-A72 both compilers display very similar performance. However, on the both Intel platforms, the LLVM compiler consistently outperforms the GCC compiler under all three execution models. However, while the data throughput of the LLVM compiler is higher on the Intel platforms, the impact of batch size on the throughputs is consistent across compilers, with the profile varying in a manner more characteristic of the platform than of the compiler.

Table 4.6: Compiler flags

Flag	GCC		LLVM	
	Intel	ARM	Intel	ARM
`-Wall`	✓	✓	✓	✓
`-pedantic`	✓	✓	✓	✓
`-fcoroutines-ts`			✓	✓
`-fcoroutines`	✓	✓		
`-std=c++2a -stdlib=libc++`			✓	✓
`-std=c++20`	✓	✓		
`-O3`	✓	✓	✓	✓

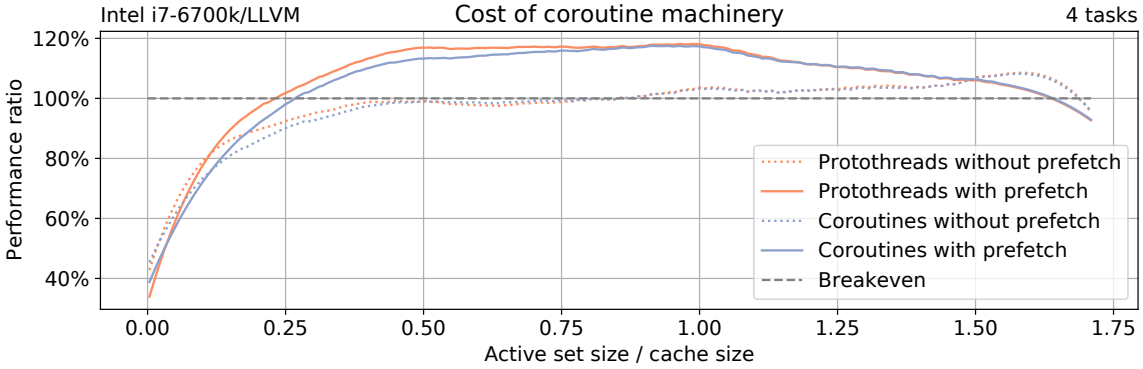


Figure 4.17: The performance characteristics of the coroutined execution models vs the Protothreads execution models. A Savitzky-Golay filter is applied to smooth the output curve. (Model: SVM; compiler: LLVM, platform: Intel 6th gen; threads: 4.)

Table 4.6 shows the flags used for each compiler on each platform. Full aggressive optimisation (`-O3`) was used for all cases. Standards were enforced: both `-pedantic` and `-Wall` were used, while `-Ofast` was avoided so as to prevent any numeric optimisations that might conflict with IEEE or ANSI standards. There are compiler-dependent variations affecting the description of C++ version, C++ library inclusion and the new instruction to include coroutine support, but the impacts of these flags are identical for all platforms and compilers.

Since `-O3` can include inlining, vectorisation, and loop unrolling for each of these compilers, the object code was inspected: no use of these features was found in the relevant sections of the test applications.

4.4.9 Coroutine machinery cost

The cost of the coroutine machinery was investigated by comparing the performance of a protothread implementation of the modified execution model with the coroutined version. The two implementations should be very similar in behaviour: the key difference is that the code for maintaining the values of local variables is hand-written in the Protothreads version. As can be seen in Fig. 4.17, the performance of the two implementations is very close across all data sizes, indicating a small and reasonably constant coroutine machinery cost. Similar results were found across all algorithms, platforms and compilers tested.

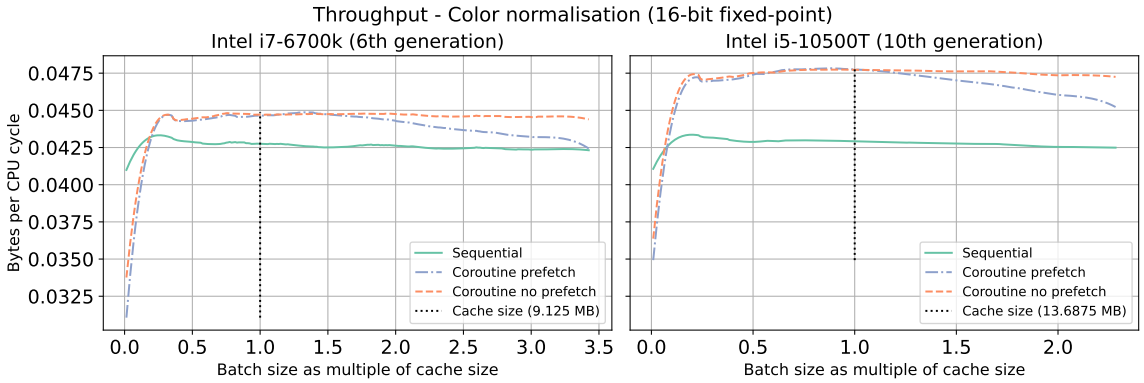


Figure 4.18: Throughput vs batch size for 16-bit fixed-point normalisation algorithms on two generations of Intel platforms.

4.4.10 Platform evolution

In general, we see an improvement in both absolute and relative performance as the Intel CPU models advance from 6^{th} to 10^{th} generation. However, some results stand out.

Fig 4.18 compares the performance on the *Norm* algorithm of two tested generations of Intel CPUs, in terms of bytes of data processed per CPU cycle. The throughput for the coroutined versions is slightly better for the 6^{th} generation CPU, and the similarity of the curves for prefetch and no prefetch indicates that the performance benefit is due to memory access patterns, rather than to explicit memory prefetching.

However, on the 10^{th} generation CPU, while the performance of the unmodified sequential execution model is substantially unchanged, the coroutined versions (with and without prefetching) display a significant improvement, resulting in a net boost of $\approx 5.5\%$. This is a windfall benefit, resulting perhaps from an architectural change in the Intel CPU cache design or memory mapping design.

4.5 Discussion

4.5.1 Performance costs & benefits

We have shown that there are some situations in which a developer can use coroutines to improve the memory access pattern, leading to significant performance improvements as shown in Fig. 4.1. Summarising the results from our work, performance boosts are likely to occur when either or both of the following factors are met: (1) there is indirection through pointers, as in the B+ tree; and/or (2) the calculation is fast so that memory latency is more limiting, as in the examples that use integer arithmetic rather

than floating point arithmetic. It will also be necessary that (3) the size of the active set can be controlled to lie within the optimal region; and (4) the CPU architecture has a sufficiently sophisticated cache hierarchy, as in the examples where the 10th generation Intel CPU performed better than the 6th generation.

The algorithm should also be structured in such a way that there is an obvious place to put a prefetch instruction and coroutine suspend. Our experiments found that there was too much overhead if the programmer needed to use an extra 'if' statement to make the decision at runtime. Therefore, there should be some part of the algorithm loop where it makes sense to include an unconditional coroutine suspend.

4.5.1.1 Alternative approaches

Other than this coroutine transformation, there are alternative approaches to achieving increased concurrency.

Techniques with low programming costs include SIMD vectorisation (enabled through the `-O3` compiler optimisation flag) and parallelism through OpenMP pragmas (e.g. `#pragma omp parallel for`) or compiler flags (`-fopenmp`). All of these offer performance improvements for the innermost loop, within the scope of a single vector operation. This is a separate strategy from the coroutine transformation, and can easily coexist with it, since the coroutine transformation is concerned with reordering the outer loops, which are not likely to be affected by the compilers' parallelisation optimisations.

Another approach is to use explicit multi-threading. As well as adding significant memory costs compared to coroutines (for the per-thread stack), multi-threading introduces potential problems with memory conflicts such as race conditions and the coding effort and complexity associated with defending against these. Finally, the use of massively parallel processor arrays such as those in graphics processor units offers far more powerful performance improvements, but at the cost of using a different language or language subset, and significantly increased hardware and power costs.

4.5.2 End-user experience

C++20 coroutines have reduced the cost of applying this method. All the hard work of transforming subroutines into coroutines, particularly maintaining local variable state efficiently and reliably, is handled by the compiler; additionally, the existence of corou-

tines as first-class features in C++20 mean that changes to the scheduling code are simpler, more robust and more likely to be reusable.

As shown in Fig. 4.3, the changes can be very simple. The possible locations for code injection are limited to the various levels of iteration: the developer selects the level that allows a prefetch of a segment of memory that is a small multiple of the cache line size (typically 64 bytes).

Our experience was that typical asynchronous code patterns for coroutining - with or without data prefetch - can be written easily and intelligibly in C++ using the new language features, adding little to the maintenance burden of the codebase. Further, we found that it was straightforward to apply the changes in a way that could be switched on or off at compile-time, providing an efficient and easily updated cost-benefit analysis of the transformations.

4.5.3 Test code

Making it possible to switch the modifications on or off at *run-time*, however, was harder work. A standard pattern for this behaviour in C++ would be to use virtual methods that differ between various sibling classes created by class factories. This approach adds significant performance cost if the virtual functions are executed (and the vtable must be inspected) in tight inner loops; therefore - for our testing purposes - it was not acceptable.

Instead, we used template classes to instantiate behaviours, so that the cost of deciding which version of a method to execute was incurred at compile-time instead of run-time. This approach has two significant disadvantages: (i) the source code can be more complex and less readable and (ii) the size of the binary executable can become significantly higher.

For these reasons we would not advise other developers to expend resources developing a *run-time* switching capability: a capability to switch these transformations on or off at *compile-time* should usually be sufficient for performance testing purposes.

4.5.4 Application scenarios

The target application of our research includes the deployment of machine-learning inference engines to low-power sensors and edge devices to support remote applications

including machinery health management and remaining useful life prediction. In this scenario, the two most successful of the algorithms tested - *B+-Tree* and *SVM* - are both applicable, and offer significant performance and battery life advantages in exchange for a very low cost in programming effort. Our model employs periodic visits to a B+ tree representation within a sensor-specific embedding and uses support vector machines to detect damaged components.

Both of these CPU-consumptive (and therefore battery-consumptive) processes can show significantly lower costs for relatively little programming effort if the coroutining and memory prefetching techniques are applied to their C++ source code.

In our tests, the lowest-powered edge device - the ARM Cortex-A72 - did not show any benefits from using these techniques for convolutional neural networks. Given how common these layers are becoming in edge devices for recognition purposes, this is a disappointing outcome. However, we note from our discussion of platform evolution that the rate of change in processor design, particularly with regard to CPU cache capabilities and heuristics, will make it worth revisiting this test in the near future as new general purpose platforms become available.

4.6 Conclusion

In conclusion, language-native coroutines make it easier for programmers to implement memory prefetching and a more efficient memory access pattern. Significant performance benefits are possible under the right conditions. We explored a comprehensive parameter space to determine the conditions under which this optimisation is beneficial.

Our work will benefit software engineers who are implementing machine learning inference on edge devices. The result will be improved performance at a minimal cost to development effort and code clarity.

Chapter 5

Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines

This chapter is an expanded and reformatted version of the following paper for the Elsevier journal *Internet of Things*.

> B. Belson, J. Holdsworth and B. Philippa, "Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines", in *Internet of Things*, doi: TBA.

The paper was submitted to the journal in April 2024, as a 25-page long-format journal article. This chapter contains extra material which was excluded from the journal version, including source code, more detailed results and some discussions of implementation issues.

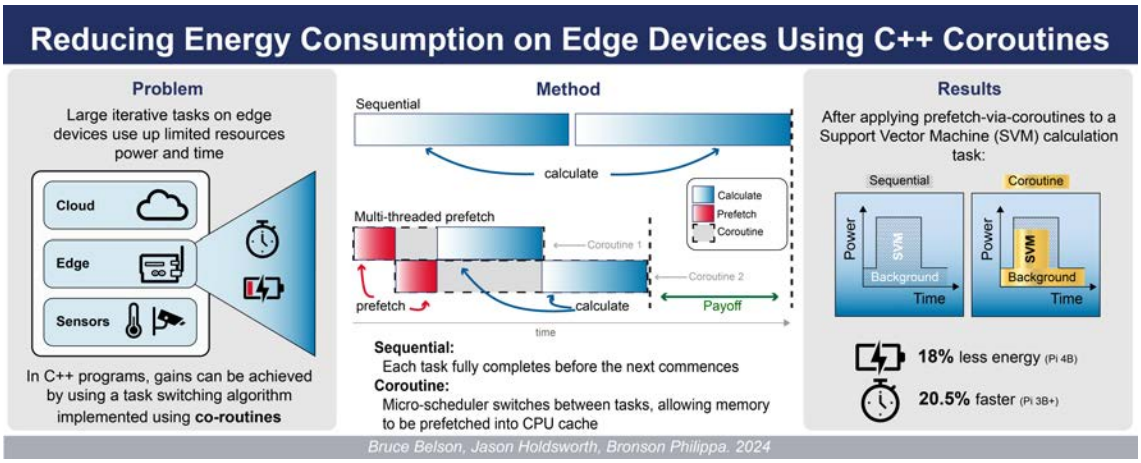


Figure 5.1: Graphical summary for chapter.

Chapter Abstract

Increasingly, machine learning inference is implemented on relatively low-powered edge devices, where battery life is a key performance criterion. In this work, the potential of C++20 coroutines to optimize the execution order of iterative inference tasks on edge devices is demonstrated. This approach is applied to a Prognostic and Health Management (PHM) application, which processes streams of vibration data as envelope spectra from a wireless sensor network using an array of Support Vector Machines. Experimental results on ARM Cortex A72 and A53 64-bit SoCs show that this method can reduce energy consumption for the task by up to 18%, reduce overall energy use by up to 20%, and cut execution time by up to 20.5%. Furthermore, peak power levels are reduced by up to 4.5%, and peak current is reduced by up to 25 mA, extending the battery life of rechargeable devices. The necessary changes to the C++ code are shown to be simple, repeatable, and broadly applicable to iterative inference tasks.

5.1 Introduction

As the volume of data produced by Internet of Things (IoT) devices grows [14, 192], an ever-increasing amount of data is processed locally, on edge devices [161, 165, 139], rather than being transmitted in full to the cloud for remote processing [139, 3]. Since many of these edge devices are battery-powered, it is important to minimise the energy consumption of machine learning (ML) inference models [160] located at the edge.

The bulk of ML inference is executed by code libraries written in C and C++. Although other languages, particularly Python, are used to control ML processing, the underlying libraries, such as those of NumPy [74], PyTorch [141] and TensorFlow [1] are primarily written in C and C++. Furthermore, inference engines designed for microcontrollers and other edge devices, such as TensorFlow Lite for Microcontrollers [39], and uTensor [163] are implemented purely in C++. Therefore, performance and efficiency improvements of C++ implementations are critical for the practical deployment of ML inference on the edge.

In 2020, the C++20 standard introduced coroutines as a native language feature [17]. Coroutines are subroutines that can be suspended and resumed without loss of local data and state [36, 119]. The C++ implementations of coroutines in LLVM and GCC execute efficiently and can therefore be used as extremely lightweight threads [85, 146, 16]. Coroutines can be used to create a "mini-scheduler" which divides large monolithic iterative tasks into smaller sub-tasks. Each of these sub-tasks can benefit from prefetching [85, 146] and improved memory access patterns [16], resulting in appreciably faster data throughput compared to the standard sequential execution pattern.

Our earlier work [16] used micro-benchmarks to investigate the impact of these techniques on the speed of execution of various algorithms employed in inference engines. However, that work was limited to micro-benchmarks and did not consider the applicability of the technique to real-world use cases, whereas this research studies the effects of a coroutine-based execution pattern on a real application and compares not only data throughput but also the energy consumption characteristics of sequential and coroutine-based execution patterns. The transformation applied to the execution pattern and its outcomes are summarised in Fig. 5.1.

The rest of the chapter is organised as follows: Section 5.2 focuses on the most recent and relevant work in the field of performance enhancement using coroutines. Section 5.3 describes the methodology of the techniques used to improve performance and the experimental methods we used to study their impact. Section 5.4 presents the results of our experiments, Section 5.5 discusses the results and Section 5.6 contains our conclusions and suggestions for possible future work. 5.7.1 lists the detailed results of the tests on each platform, and 5.7.4 contains the relevant portions of the experimental source

code. The code for the experiments can be downloaded from an online repository[1].

5.2 Related work

The move towards edge processing of IoT data is driven by several different factors, including on the one hand data-related concerns such as privacy and security [198] and data provenance [78], and on the other hand network-related issues such as latency [53, 150, 29, 3], reliability [53, 29, 112] and bandwidth [112].

The problems and mitigations of memory access patterns have been examined in the context of ML on edge devices [12, 22]. Numerous approaches have been explored, including hardware [56, 31, 183, 86], software [191, 195] and hybrid [183, 22] perspectives. Processing-in-memory (PIM) has been proposed as a means of improving speed and reducing unnecessary data movement during local processing of edge data [59, 65], and supporting architectures are now commercially available [102, 104].

Now that they are part of a language standard, C++ coroutines can delegate the work of implementation to the compiler: thus the trade-off between code complexity and performance has changed – to the advantage of the developer. Techniques based on coroutines have been used to improve the performance of database engines on large server platforms [75, 146, 85] and to implement software-defined networking [8].

On a Raspberry Pi 4 B – a small platform widely used for edge processing – coroutines were used to implement a "mini-scheduler" that provided up to 65% speed improvements for a range of ML algorithms and transformations, including 8.3% speed improvement for a Support Vector Machine (SVM) [16].

5.3 Methodology

5.3.1 Application

Following on from the successful application of coroutines to repeated SVM execution on edge devices [16], a real-world application was selected that seemed likely to benefit from the same treatment: the application runs on an edge device and performs multiple SVM calculations across multi-layered data sets.

[1] https://github.com/bbelson2/coro_edge_energy.git/

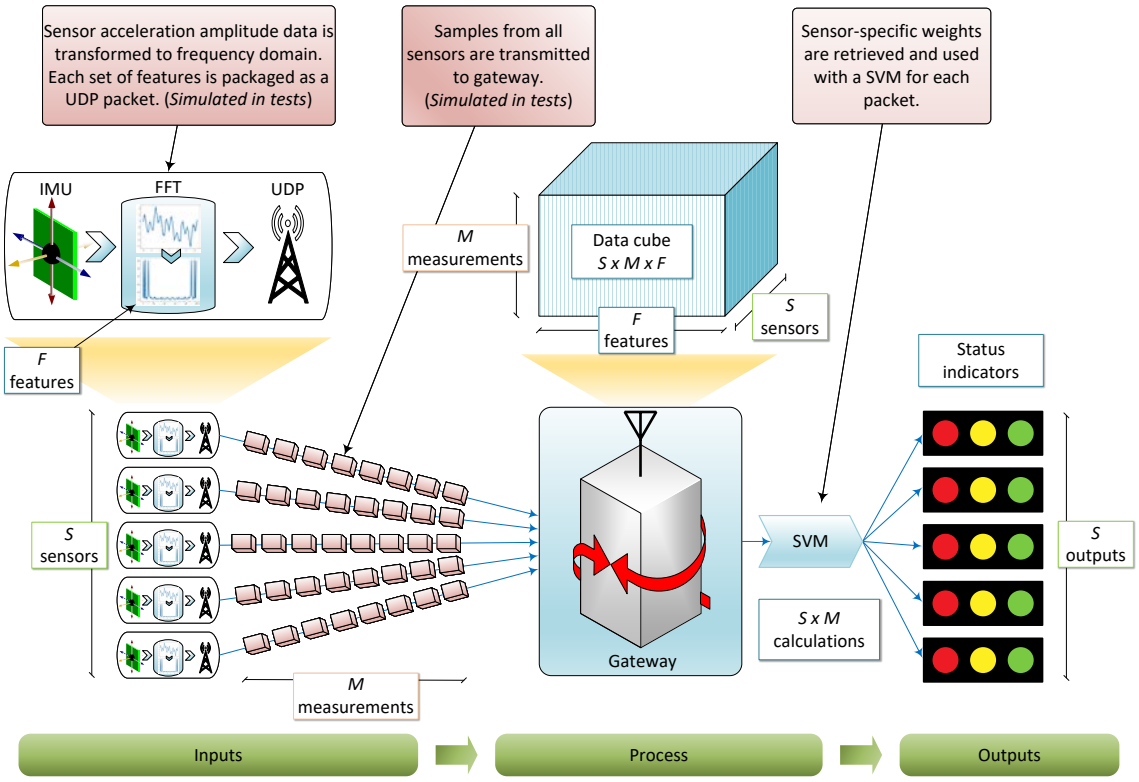


Figure 5.2: Networked application. (i) Sensor acceleration data is recorded by inertial measurement unit (IMU) and transformed to frequency domain; (ii) sample vector is sent to gateway as a UDP packet; (iii) gateway retrieves sensor-specific weights and applies SVM to each sample vector.
S sensors each produce M measurements, each containing F features; thus input data is very large ($SxMxF$), but the final output – just one status value for each sensor – is small (S). This architecture is characteristic of edge systems in remote deployments without easy access to the cloud: it avoids cloud dependency, reduces cost and assists privacy. The architecture requires that inference be implemented locally; inference involves many layers of iteration of simple calculations.

The studied architecture is shown in Fig. 5.2. We consider the case of machine learning inference running on a gateway attached to a wireless sensor network (WSN). Each WSN node reports on the health of a machine. The sensor node uses an accelerometer to record the vibrations of the machinery for a sample period, then converts the data to the frequency domain using a locally executed Fast Fourier Transform (FFT). The FFT spectrum is then transmitted to the gateway where the Prognostic and Health Management (PHM) method is applied as follows: the latest spectrum is compared against a set of expected frequencies for the specific machine using a SVM.

The gateway application receives data transmitted by the S sensors in the WSN. Each measurement contains a vector of F FFT bins, representing a set of F features: it is packed into a single UDP datagram and transmitted to the gateway. M measurements

are collected for each sensor. When all data has been received, the gateway processes all measurements for all sensors. Finally, a single status value for each sensor is transmitted to the cloud.

This architecture – with large amounts of input data and small quantities of output – is typical of many applications: local processing avoids cloud dependency, reduces communications costs and assists privacy. In cases such as this one, where the application is deployed remotely, cloud solutions are not feasible.

Each vector of features within the complete collection of measurements is passed through a Support Vector Machine (SVM) using per-sensor weights. This calculation is a multi-level iteration process as described in Algorithm 1.

Algorithm 1 Levels of iteration within the application

```
1:  procedure SVMs(sensor_data)          ▷ Apply SVM to each feature vector in sensor_data
2:      for all s in sensor_data do          ▷ Iterate across sensors - S items
3:          measurements ← measurements[s]      ▷ Retrieve all measurements for this sensor
4:          weights ← weights[s]                 ▷ Look up weights for this sensor
5:          for all m in measurements do     ▷ Iterate across measurements - S x M items
6:              features ← m                     ▷ m contains a vector of features
7:              ▷ Injected code: Initiate a prefetch for [features] ◁
8:              ▷ Injected code: Suspend execution until [features] is loaded into cache ◁
9:              total ← 0
10:             for all f in features do     ▷ Iterate across features - S x M x F items
11:                 total ← total + f.w[i]       ▷ Calculate step of dot product
12:             results[m] ← (total > bias[s])   ▷ Calculate class of this measurement
13:         outcomes[s] ← any(results)           ▷ Calculate outcome for this sensor
14:     return outcomes                          ▷ Return outcomes for all sensors
```

The SVM calculations for a complete data set provide a test for the coroutine-based execution pattern which this work examines. This multi-layered set of iterations offers an opportunity for a simple transformation to parallelised, multi-threaded execution, using coroutines as extremely light-weight threads, under the model shown in Fig. 5.3, as demonstrated in earlier work [16].

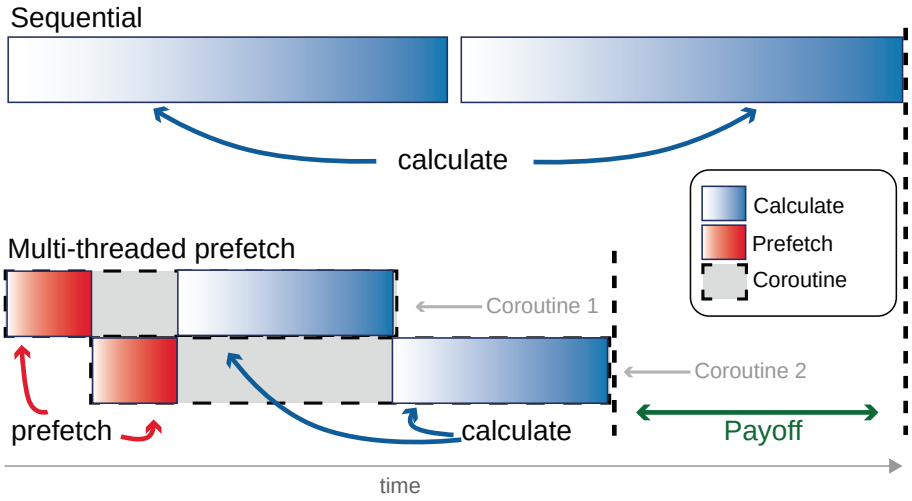


Figure 5.3: Coroutine execution model compared with unmodified sequential execution model. (Based on [16] with permission.)

5.3.2 Platform

The gateway computer program was authored in C++20, the first version of C++ to support coroutines [80]. The Clang 12.0 compiler was used, based on its efficient implementation of coroutines [16]. The program was executed on a Raspberry Pi 4 computer [181] with 2GB RAM and a quad core ARM Cortex A72 64-bit SoC with a two layer CPU cache. The L1 cache size of this machine is 32 KiB and the L2 cache is 1024 KiB. 16-bit fixed point numbers (with 3.13 bits) are used for all real numeric values. A secondary platform - the older Raspberry Pi 3 B+ - was also tested for comparison. (We restricted our focus to the Raspberry Pi ecosystem (i) for simplicity and (ii) as a good fit for available test equipment.) The platforms are summarised in Table 5.1 and in Table 5.7 in the appendix section.

5.3.3 Coroutine implementation

The coroutine transformation summarised in Fig. 5.3 and in Algorithm 1 improves the speed of iterative tasks that access arbitrarily located blocks of memory: in the case of the SVM operations in the test application, an earlier isolated micro-benchmark [16] demonstrated performance improvements of up to 8.3% on this platform.

Some small modifications to the SVM implementation code are required before it can make use of the coroutine execution model:

1. The function containing iterations must be transformed to a coroutine. This is

Table 5.1: Software characteristics

Name	Value	Notes
C++ version	C++20 [80]	Earliest C++ version to support native coroutines; only C++ version fully implemented at time of testing
Compiler	Clang 12.0	Most efficient coroutine implementation for this platform [16]
Operating system	Debian Linux 11 (Bullseye)	Minimal implementation (Raspberry Pi OS Lite)
Arithmetic	16-bit fixed-point numbers	3 integer places and 13 fractional places were sufficient for the range and resolution of the data set, which is derived from 12-bit accelerometer readings

done by (i) returning an object that implements the `promise_type` interface and (ii) calling `co_await`, `co_yield` or `co_return` at some point.

2. Within the iterator two operations must be injected before each new data region is used: (i) initiate a prefetch for the data and (ii) yield to the scheduler by calling `co_await`. Fig. 5.16a shows the C++ code of the simple modification required for the application code, which is inserted before each SVM operation.

As a result of these changes, the coroutine executing the SVM pauses while the required data is loaded into CPU cache. Loading into cache is executed asynchronously by dedicated machinery independent of the CPU. The mini-scheduler ensures that another task is executed while the cache is loaded. When the scheduler again invokes the waiting task, the data will be in cache and no stall will occur. Furthermore, speed of execution also benefits from an advantageous memory access pattern [16].

5.3.4 Execution template

The execution context is controlled by a mini-scheduler, implemented as a C++ template. The template is reusable across any multi-level iterative task and is shown in Listing 5.1. In order to be called by the scheduler, the coroutine that executes a single step must be reorganised so that it accepts the following parameters: (i) a reference to an application-dependent context class and (ii) the current index into the collection of items to be iterated. This allows the scheduler to manage the list of remaining work efficiently, and without any knowledge of the specific domain or task. An example of

this reorganisation is shown in Listing 5.2.

An instance of the `coroutine_runner` templated class is instantiated and then invoked via its `run()` method (as shown in Listing 5.3). The template runs as defined in Algorithm 2, summarised as follows:

1. The `run()` method creates an collection of coroutines - one for each parallelised lightweight thread.

2. The method then repeatedly iterates through the collection of coroutines in a round-robin pattern. If a coroutine's current work item is in a wait state it is resumed. If the item is complete then the coroutine is deleted and replaced by a new one, created for the next item in the queue.

3. The coroutine currently being executed may pause at any time and enter a wait state by calling `co_await`; control then returns to the scheduler's `run()` method, which selects another waiting coroutine.

4. Once all items have been completed, the `run()` method exits.

5.3.5 Test application

A new application was derived from an existing application specifically to support the test. For the purposes of repeatability, the sensor nodes were simulated during this testing (using a pseudo-random series of feature values from a Mersenne twister, MT19937, with a fixed seed), so as to generate repeatably the UDP traffic representing the sensor network. This study focused on the gateway part of the WSN and specifically on the performance characteristics of the numeric processing required for the local machine learning inference engine.

In addition to the use of simulated data, the code was altered as follows: the two execution models (the standard unmodified sequential model and the modified coroutine-based model) were run alternately for each set of data, in an interleaved pattern; additionally, large regions of memory were modified between each test, in order to flush the CPU cache, and to ensure that each test started without any relevant memory already in cache.

The execution models were run repeatedly as follows:

1. Receive and collate multiple UDP datagrams from a number of sensors (simulated), then flood cache.

2. Run test 30 times:

   (a) Apply SVM to each data set using standard sequential execution, then flush cache.

   (b) Apply SVM to each data set using coroutine execution, then flush cache.

The execution parameters were varied as shown in Table 5.2, for a total of 1,550 tests, each test being run 30 times.

Table 5.2: Execution parameters

Parameter	Symbol	Range	Step	Description
Sensors	S	10 - 50	10	Number of sensors transmitting data packets
Measurements	M	10 - 100	10	Data packets per sensor
Features	F	128 - 2048	64	Number of FFT bins (features) per measurement
Repeats		30		Number of repeats for each test

5.3.6 Performance measurement

The power supplied to the Raspberry Pi was measured using a Joulescope 220 [2], as shown in Fig. 5.4. The Joulescope recorded execution time and power usage with resolution of 0.5 μs and 875 μW respectively.

The test application set and cleared general-purpose input/output (GPIO) pins on the Raspberry Pi at the start and end respectively of each processing phase, and this data was passed to the Joulescope to synchronise with the energy measurements, and to be collected in the same result set, as shown in Fig. 5.5.

It could be argued that this study might have been more effectively executed by using a simulation of the platform, rather than through the actual time and power measurements used in this work. However, we decided that building confidence in the actual outcomes – especially with regard to battery life and speed of performance – was important enough to justify the additional time and effort.

[2] https://www.joulescope.com/

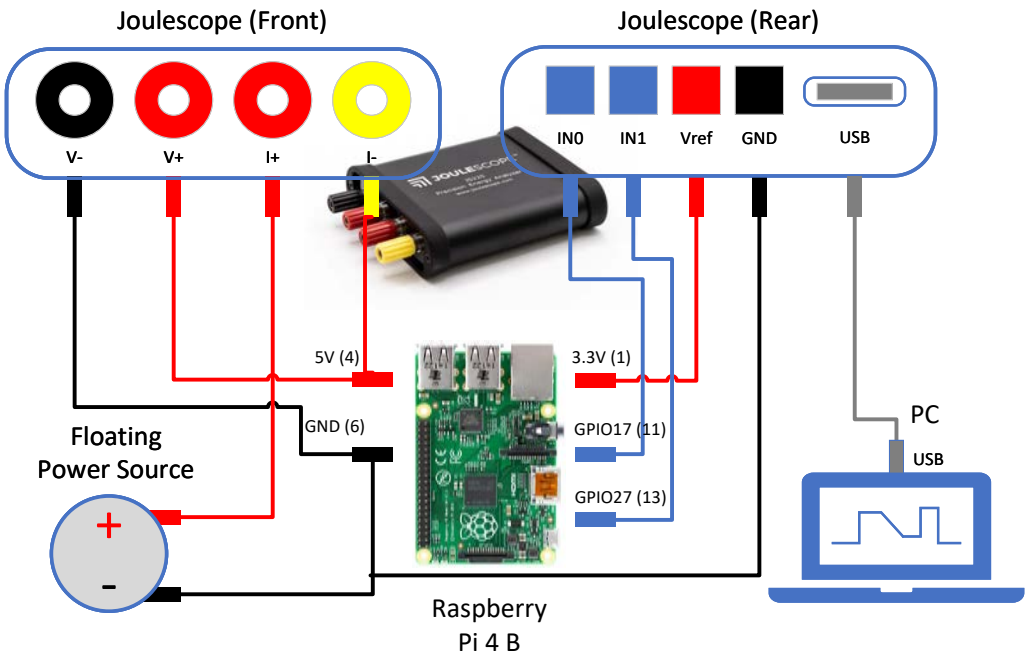


Figure 5.4: Wiring layout for experimental procedure. The Joulescope provides power from the floating source to the Raspberry Pi, and measures voltage & current used. The Raspberry Pi provides timing information to the Joulescope through interrupts. All data is merged and passed to a PC via USB.

5.3.6.1 Time Saving

Time (T) measures the time from the start of the SVM calculations to the end, as shown in Eq. (5.1). T is the simplest result to measure, since it can be read immediately from the width (along the time-axis) of the rectangular pulse from the appropriate GPIO pin, as shown in the lower two traces in Fig. 5.5.

$$T = t_1 - t_0 \tag{5.1}$$

$$\text{where } t_0 = \text{start time},$$

$$t_1 = \text{end time}.$$

Tests are examined in pairs: the standard sequential execution and the coroutine execution pattern that follows it. The Time Savings (S_T) for the pair is calculated as shown in Eq. (5.2). Thus a positive value signifies an improvement in performance (i.e. less time used for the same task).

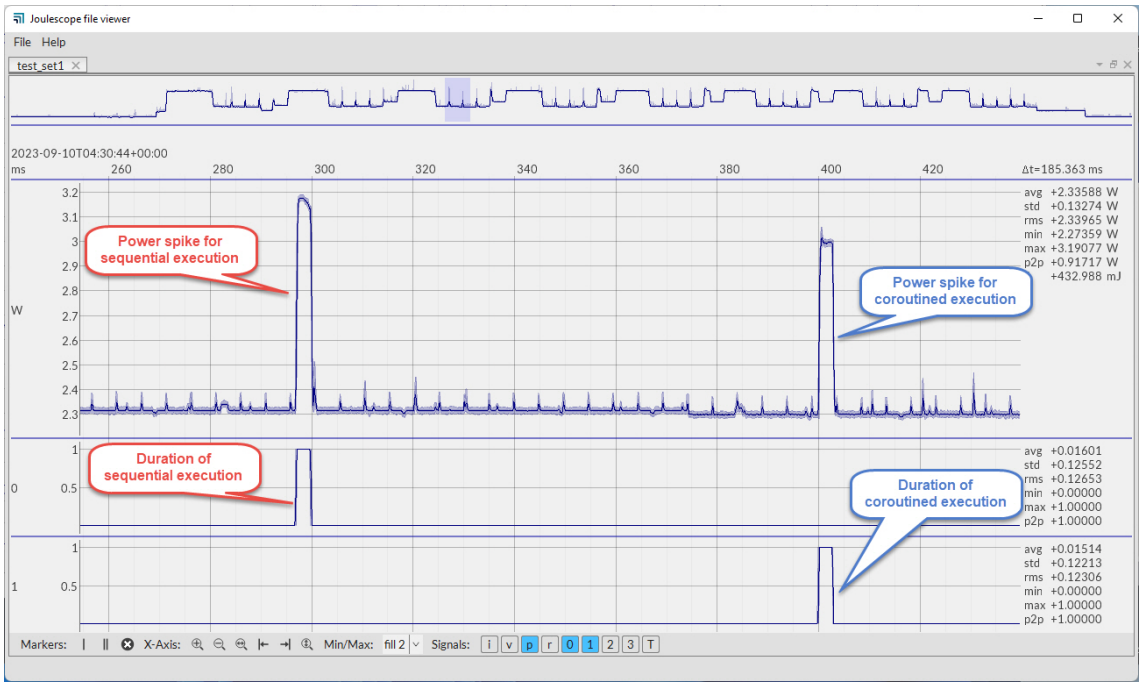


Figure 5.5: Example readout from Joulescope, showing the power usage of the device under test (in the upper trace) along with the GPIO pins (in the lower two traces), indicating which execution context is in use.

$$S_T = \frac{T(S) - T(C)}{T(S)} \tag{5.2}$$

where $T(S)$ = duration of sequential pattern,

$T(C)$ = duration of coroutine pattern.

5.3.6.2 Overall power and energy savings

Overall Energy (OE) measures the total energy used by the test device from the start of the SVM calculations to the end, as shown in Eq. (5.3). OE is simple to measure, being the area under the power curve (the upper trace in Fig. 5.5) between the start and end of the duration's rectangular pulse.

$$OE = \sum_{t_0}^{t_1} P.\Delta t \tag{5.3}$$

where P = power measured at Joulescope,

Δt = time resolution of Joulescope.

The energy used by each pair of tests is measured. The Overall Energy Savings (S_{OE}) for the pair is calculated similarly to duration savings, as the reduction in energy divided by the energy of the sequential pattern, as shown in Eq. (5.4). A positive value signifies an improvement - a reduction in energy used.

$$S_{OE} = \frac{OE(S) - OE(C)}{OE(S)} \tag{5.4}$$

$$\text{where } OE(S) = \text{total energy for sequential pattern,}$$

$$OE(C) = \text{total energy for coroutine pattern.}$$

Median Overall Power (OP) for each test is calculated as shown in Eq. (5.5):

$$OP = median([P(t_0)..P(t_1)]) \tag{5.5}$$

$$\text{where } P = \text{power measured at Joulescope,}$$

$$t_0 = \text{start time,}$$

$$t_1 = \text{end time.}$$

The Savings in Median Overall Power (S_{OP}) for the pair is calculated as the negative of the change in power use divided by the power use of the sequential pattern, as shown in Eq. (5.6). A positive value signifies an improvement - a reduction in median power used.

$$S_{OP} = \frac{OP(S) - OP(C)}{OP(S)} \tag{5.6}$$

$$\text{where } OP(S) = \text{median total power for sequential pattern,}$$

$$OP(C) = \text{median total power for coroutine pattern.}$$

5.3.6.3 SVM task energy

As observed above, Time (T) and Overall Energy (OE) are straightforward to measure; however, the power used *specifically* for the SVM task is more difficult to derive.

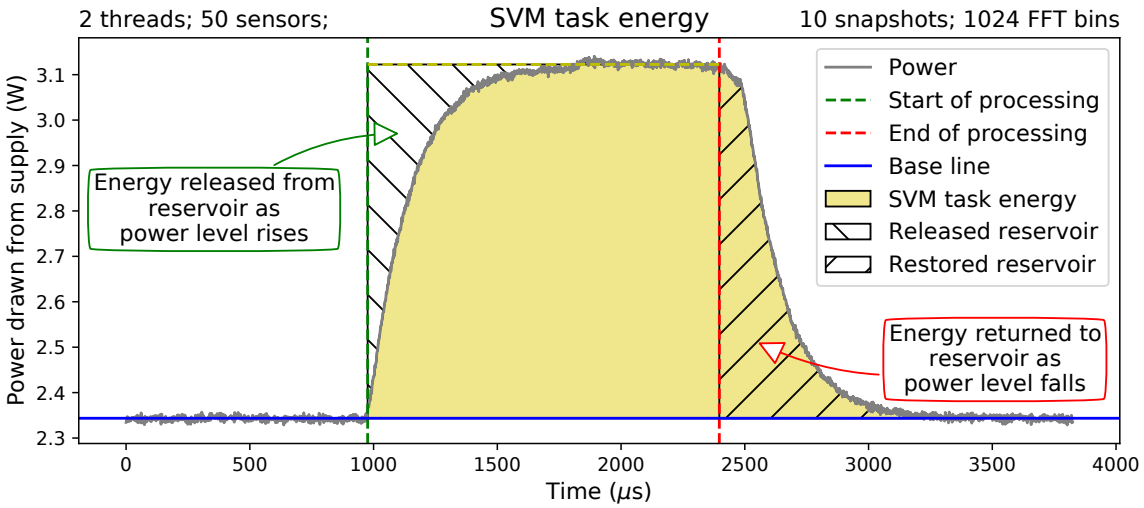


Figure 5.6: The pattern of power use for the SVM calculations. Note the delayed rise in power level at the start of processing, due to the release of energy from capacitive reservoirs and the delayed fall after the processing is complete, as the reservoirs are refilled.

Each period of SVM processing caused an immediate rise to a higher power usage level, as shown in Fig. 5.6. There was a slight delay in reaching the higher level; the length of the delay was consistent, and independent of the duration of the SVM calculations. We attribute this to a discharge of capacitors or other energy storage elements, where the additional power draw was temporarily taken from the capacitors until the voltage regulator was able to respond. There was a similar delay in returning to base power levels at the end of processing, which we attribute to the recharging of the capacitors. The process is described in Fig. 5.6.

To account for the capacitive-based rise and fall of the measured power, we define the SVM's Task Energy (TE), which is illustrated by the shaded region in Fig. 5.6. The mathematical definition is as follows.

Starting with a Base Line Power Level (P_{base}) calculated as the median power level in a fixed period before the test begins, as in Eq. (5.7), we calculate the excess of power use over base line from the start of the calculation and to the point where power use returned to the base line, as shown in Fig. 5.6 and in Eq. (5.8).

$$P_{base} = median([P(t_0 - t_\delta)..P(t_0)]) \tag{5.7}$$

where P = power measured at Joulescope,

t_0 = start time,

t_δ = discharge period.

$$TE = \sum_{t_0}^{t_1+t_\delta} (P - P_{base}).\Delta t \tag{5.8}$$

where t_1 = end time.

As with Overall Energy, the SVM's Task Energy Savings (S_{TE}) are calculated as the ratio of the reduction caused by the use of coroutines and the original SVM Task Energy, as shown in Eq. (5.9).

$$S_{TE} = \frac{TE(S) - TE(C)}{TE(S)} \tag{5.9}$$

where $TE(S)$ = SVM task energy for sequential pattern,

$TE(C)$ = SVM task energy for coroutine pattern.

5.3.6.4 Peak power

Peak current – and thus also peak power usage (PP) – is known to have an effect on total battery lifetime [27]. The peak power level during the SVM processing was calculated as shown in Eq. (5.10).

$$PP = \max_{t_0 \to t_1}(P) \tag{5.10}$$

where P = power measured at Joulescope,

t_0 = start time,

t_1 = end time.

The savings in peak power level (S_{PP}) were calculated by comparing peak power usage for each test pair as shown in Eq. (5.11).

$$S_{PP} = \frac{PP(S) - PP(C)}{PP(S)} \tag{5.11}$$

where $PP(S) =$ peak power for sequential pattern,

$PP(C) =$ peak power for coroutine pattern.

5.3.7 Statistics

Each test was repeated 30 times in order to provide a large enough test set to detect outliers (based on a rule-of-thumb emerging from the Central Limit Theorem). Although the operating system version was "headless" and thus had a minimised number of competing processes and daemons, Raspberry Pi Linux is nevertheless a multi-tasking operating system, and performance of any task will inevitably vary, impacted by the heightened activity of background processes.

Median absolute deviation (MAD) [158] was used to detect outliers. Within each test, the MAD of the duration was calculated for each execution pattern. Results whose deviation from the MAD was more than 4 standard deviations (above or below) were excluded. For any pair of results (i.e. consecutive sequential and coroutined executions), the exclusion of either result resulted in the exclusion of both.

Surviving pairs of results were retained, and for each pair the changes in performance for time, energy used, median power used, adjusted energy used and peak power level were calculated as described in Eqs. (5.2), (5.4), (5.6), (5.9) and (5.11) respectively.

This process resulted in the removal of 8.74% of the test runs for the Raspberry Pi 4 and 15.93% of the Raspberry Pi 3 test runs.

5.3.7.1 Outlier removal

The value of 4 for the exclusion barrier was based on an exploration of the impact of interference by other processes on the test process. Fig. 5.7 shows a selection of experimental data sets. It can be observed that true outliers (shown as red crosses in the figure) lie well beyond the barrier for 4 standard deviations. However, in sets that have zero or one true outlier, a number of non-outlier points lie close to or outside of the barrier for 3 standard deviations and might be excluded if the conventional value of

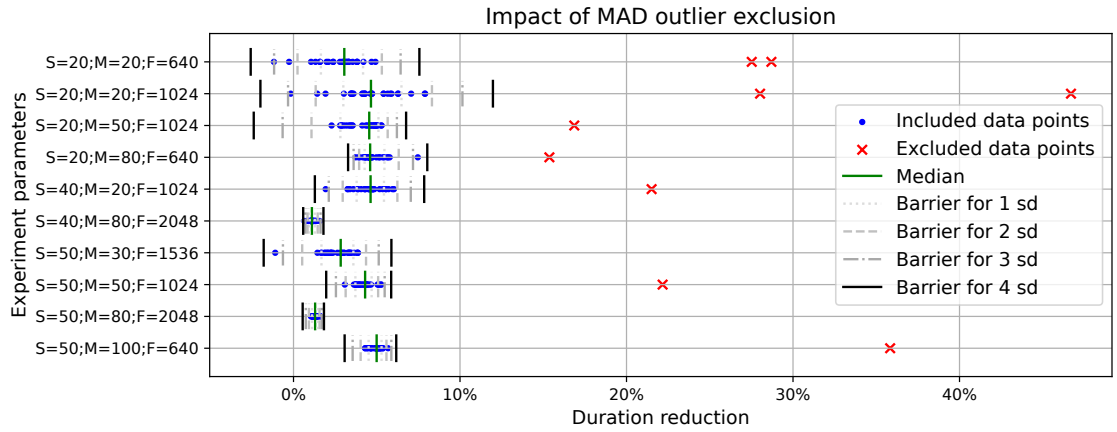


Figure 5.7: The distributions of a selection of experimental result sets, and the barrier levels for MAD exclusion at 1, 2, 3 & 4 standard distributions for each set. The Y axis tick labels show the experiment parameters (sensor count (S); sample count (M); datagram size (F)). The vertical lines show the median and the various barrier levels for each experiment. Included data points are shown as blue dots and excluded data points are shown as red crosses. Notice that if an experiment has outliers, they lie well beyond the barrier for 4 standard deviations. However, for some experiments with zero or one true outliers, a number of non-outlier points lie close to or outside of the barrier for 3 standard deviations.

3 standard deviations were used.

It can be seen from Section 5.4.1.1 that this approach excluded 50% or more of the samples of an experiment in only 0.6% of cases. All of these low surviving sample sizes were located at the edge of the explored parameter space, with the lowest values for SxM (sensors x samples).

5.4 Results

5.4.1 Introduction

A typical output from the experiment resembles Fig. 5.5. The lower two lines on the chart display the rectangular signals supplied directly to the Joulescope by the test application, using the test device's GPIO pins: high signals on pins 0 and 1 signify the normal sequential execution pattern and the coroutined execution pattern respectively. The upper line records the power usage of the test device on the same time axis.

It is immediately apparent – from inspection of test runs such as that in Fig. 5.5 – that the spike in power use during the coroutined execution is lower than that during sequential execution. Closer examination of the rectangular pulses reveals that the total

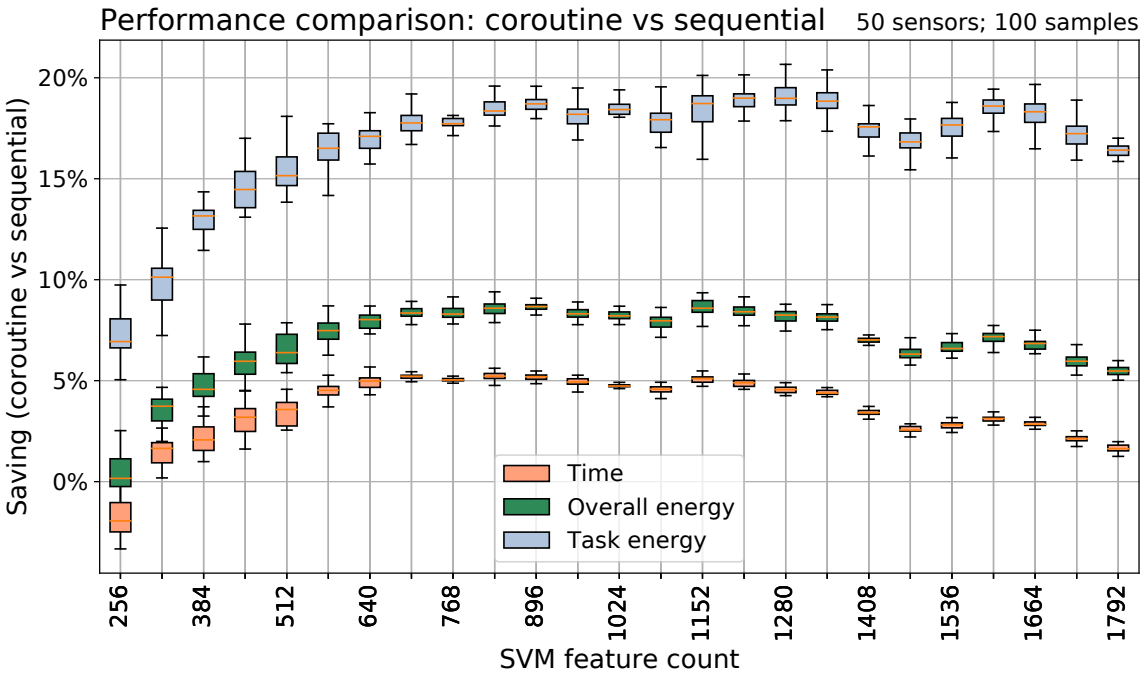


Figure 5.8: Summary of performance gains from using coroutines to reorder execution of SVM analyses on Raspberry Pi 4. Statistics displayed represent the reduction in cost divided by the original cost. **Time** is the elapsed time spent on the SVM task; **Overall energy** is the total energy consumed by the edge device during the task; **Task energy** is the marginal energy consumption of the task, after subtracting the base energy use.

time for the coroutined operation is also noticeably less than that for the sequential execution.

A typical response to the application of coroutines to the performance in terms of time and energy is summarised in Fig. 5.8. In this configuration, the use of coroutines resulted in approximately a 5% saving in time and a 19% saving in task energy consumption.

5.4.1.1 Outliers

Table 5.3 summarises the effect on sample sizes of removing outliers: the bulk of tests (89.3%) were reduced by 17% or less, i.e. no more than 1 in 6 results were identified as outliers. A very small number of tests (10 out of 1550, or 0.6%) had sample sizes reduced by more than 50%.

Sections 5.4.2 to 5.4.7 examine the savings measures individually. Tables 5.5 and 5.6 contain the summary statistics of the results for each sensor count and measurement count combination.

Table 5.3: Outliers and survivors

Survivors		Outliers		Tests	
Count	%age	Count	%age	Count	%age
28-30	93-100%	0-2	0-7%	956	61.7%
25-27	83-90%	3-5	10-17%	428	27.6%
22-24	73-80%	6-8	20-27%	93	6.0%
19-21	63-70%	9-11	30-37%	45	2.9%
16-18	53-60%	12-14	40-47%	18	1.2%
13-15	43-50%	15-17	50-57%	5	0.3%
10-12	33-40%	18-20	60-67%	5	0.3%

5.4.1.2 Dimensionality

Many of the figures in this chapter, including Figs. 5.8, 5.9, 5.10 and 5.13, focus on the impact of SVM feature count on performance improvements. SVM feature count in this set of experiments is important in that it represents a class of parameter that cannot easily be manipulated as a configuration setting or a calculated run-time setting, because it is the fixed size of an indivisible unit of work. In this case it represents the length of a feature vector input to the SVM, whose size is a side-effect of the accelerometer used in the sensors.

By contrast, the sensor count (S) and the measurement count (M) represent parameters that *can* be controlled – or tuned – by the application developer; this is similar to a Deep Learning scenario where the developer tunes the batch size in order to optimise the learning task.

5.4.2 Time

It was expected that SVM feature count (F) would have a major impact on all types of savings, since the cost of pausing and resuming the iterator with `co_await` is incurred exactly once for each feature vector, and this cost is non-trivial.

Fig. 5.9 confirms this expectation: it shows the impact of SVM feature count on time savings (S_T) for a range of sensor counts (S) and measurement counts (M). It is notable that the pattern of the response with regard to feature count (F) is consistent for almost all sensor counts and measurement counts: a small feature count results in zero time savings; as the size of the vector of features increases the savings increase dramatically,

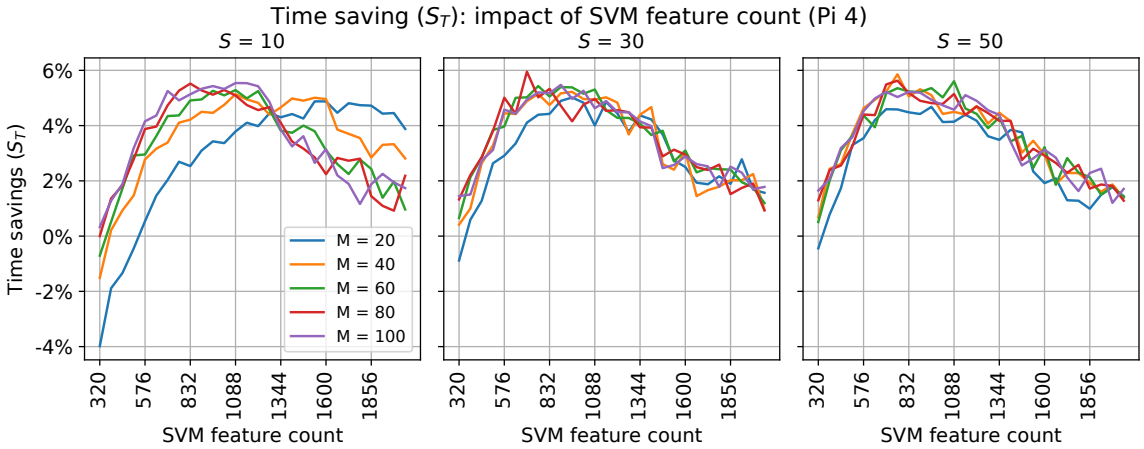


Figure 5.9: The effect of SVM feature count on time savings for various sensor counts (S) and measurement counts (M). Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 5.3.7. Note the similarity of the response curves across all sample counts. The notable exception is for the smallest sensor and measurement counts (10 x 20 and 10 x 40) which - with a smaller total memory requirement - display less decay for higher feature counts.

reaching a peak of 5.5% at a feature count of around 960-1088; the savings gradually and steadily fall off thereafter. There is an exception to the pattern for the smallest sensor x measurement count shown (10 x 20), presumably reflecting a total memory size which does not fill the CPU cache.

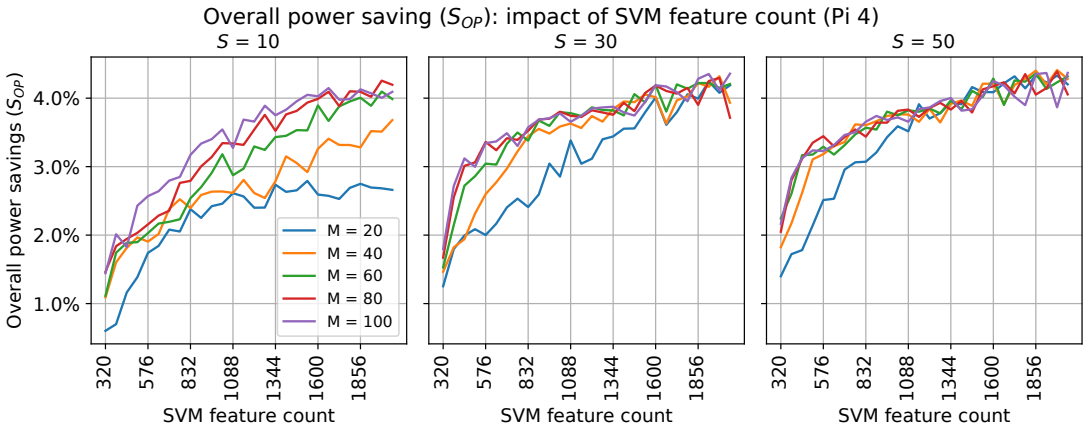
In summary, there exists - for all except the smallest network sizes - a large region of SVM feature counts where the benefits of the coroutined execution pattern are guaranteed and offer between 4% and 5.5% savings in execution time.

These results are consistent with the time savings found for SVM on a Raspberry Pi 4 B in [16].

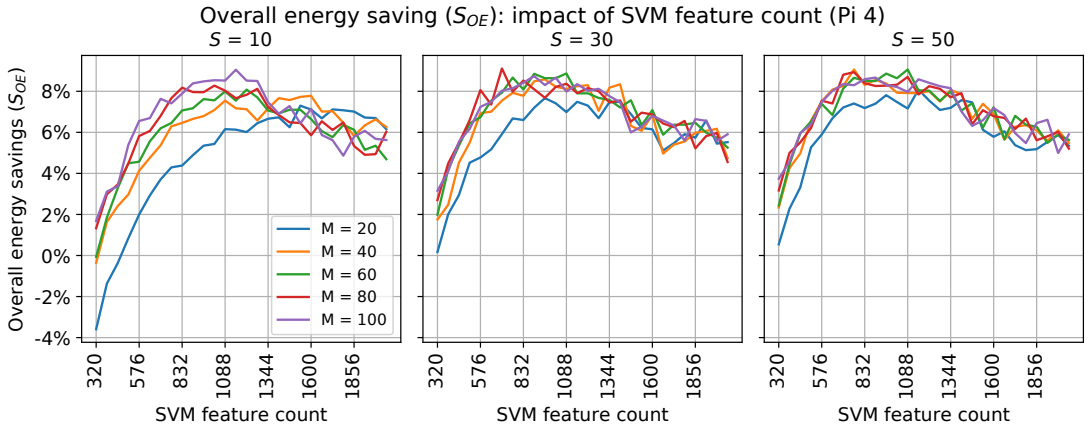
5.4.3 Overall power and median overall energy

A notable outcome of this research is the clear saving in energy usage as a result of replacing the sequential execution pattern with a coroutined execution pattern. Fig. 5.10a shows the savings (S_{OP}) in overall power usage (OP) between the two execution patterns for a number of different data set sizes and SVM feature counts. Under the modified (coroutined) execution pattern the calculation runs for a shorter time and uses less power while running; outside the smaller feature count zone, the amount of power saved rises steadily, to between 4% and 4.5%.

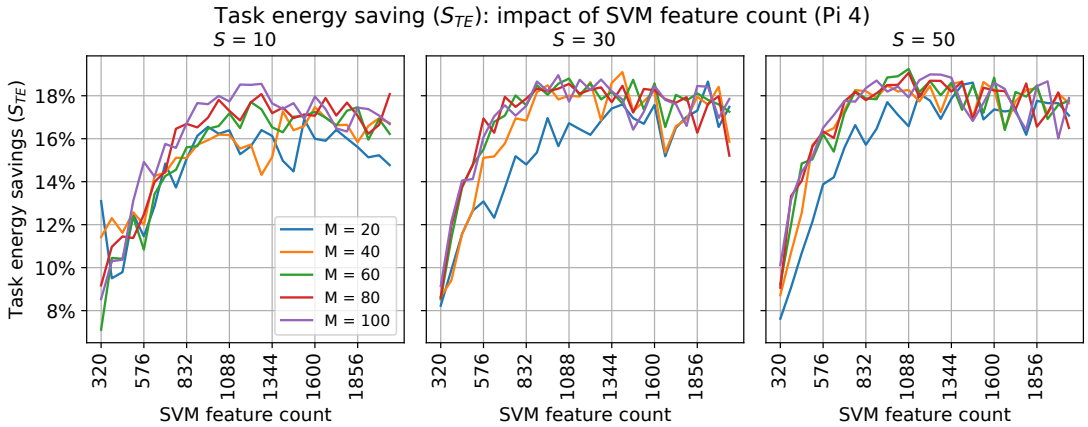
As shown in Fig. 5.10b, the effects of time saving and power saving combine to create



(a) The effect of SVM feature count on overall power savings. The chart shows power savings across all feature counts, rising steadily with feature count, and reaching a plateau at a feature vector size of about 2000.



(b) The effect of SVM feature count on overall energy savings. The effect is similar across all measurement counts except for the data sets with the smallest number of steps in each separate processing task (i.e. 20 measurements).



(c) The effect of SVM feature count on SVM task energy savings for various sensor counts.

Figure 5.10: The effect of SVM feature count on energy & power savings measures, for various sensor and sample counts. Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 5.3.7.

an overall energy saving (S_{OE}) between 6% and 9%: this saving applies to a usefully large range of feature counts. As would be expected, the impact of feature count on S_{OE} shows similar characteristics to its impact on S_T. The highest energy savings appear for feature counts between 720 and 1024. Once again, there is a consistent shape of response curve for all sensor and measurement counts, and once again there is a notable exception for the smallest sensor and measurement count (10 x 20) which also displays less decay for higher feature counts.

5.4.4 SVM task energy

This derived measure represents the energy savings specifically for the SVM task (as described in Section 5.3.6.3 and in Fig. 5.6). Fig. 5.10c shows steady energy savings of over 16% for a wide range of SVM feature counts. The measure follows a similar pattern to the overall energy shown in Fig. 5.10b. Tests with a feature count between 720 and 1024 show the highest savings, but with a very gradual falling off above 1024.

Fig. 5.10c shows that the highest range of task energy savings (i.e. $>=$ 16%) coincides with positive time savings (0.5% to 6%) and high total energy savings (5% and above). We observe that there exists a large and reliable range of data sizes where speed, total energy and task energy can all be reliably improved by use of the techniques outlined here.

5.4.5 Peak power

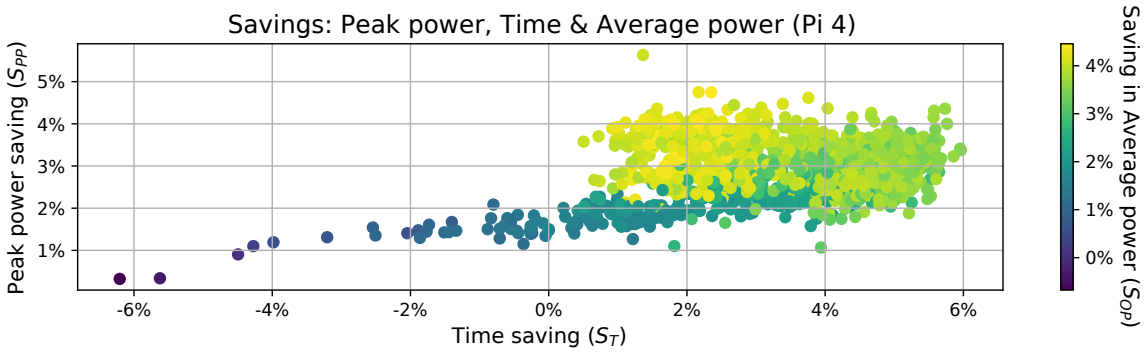


Figure 5.11: Comparison of peak power savings with savings in time and average power for the Raspberry Pi 4 B. Each point represents the results for a pair of parameter values: S (sensor count) and M (measurement count). Points to the right of 0% indicate time savings; points that are lighter in colour indicate savings in average power. Notice that the parameter values for S and M which result in useful outcomes for time and power also display useful reductions in peak power – between 2% and 4.5%.

Fig. 5.11 compares the median peak power savings for each test set with savings in time and average power for the Raspberry Pi 4 platform. The regions of parameter space which display the best average power and time savings – i.e. the brighter dots (shown in yellow and pale green) on the right-hand-side of the chart – also exhibit steadily positive peak power savings of between 2% and 4.5%.

Since peak power levels were between 2.66 W and 3.33 W with a 5V power supply, this saving typically reduced peak current from e.g. 545 mA to between 531 and 520 mA, a saving of between 14 and 25 mA.

There were no appreciable or consistent peak power savings for the Raspberry Pi 3 B+ platform: the average peak power level was in fact slightly higher on the older platform, as shown in Fig. 5.12.

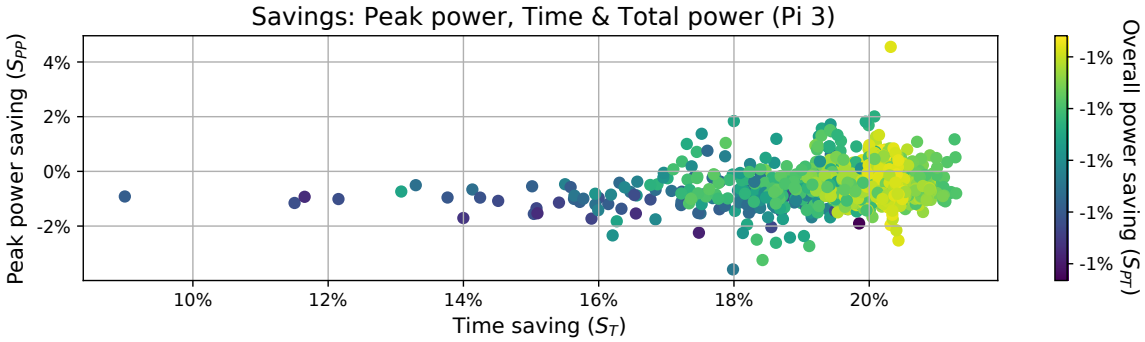


Figure 5.12: Comparison of peak power savings with time and total power savings for the Raspberry Pi 3 B+.

5.4.6 Comparison with Raspberry Pi 3

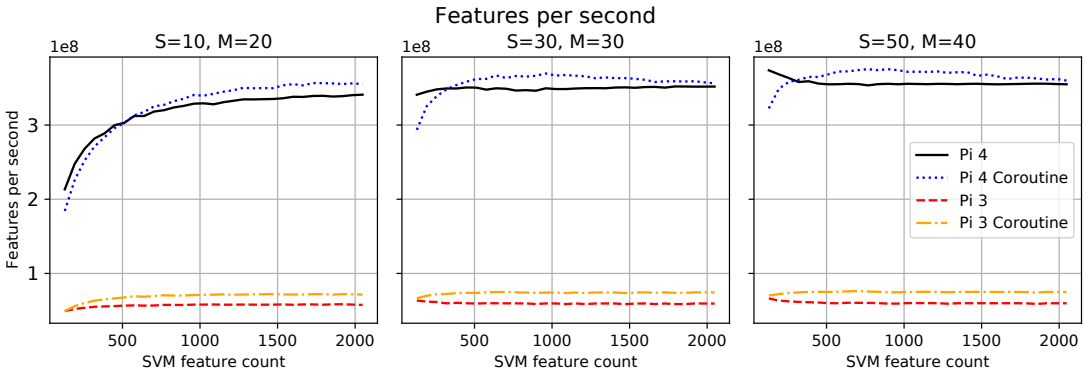
To compare base performance across platforms, we used the following calculated statistics:

$$\text{Features per second} = \frac{S * M * F}{T} \tag{5.12}$$

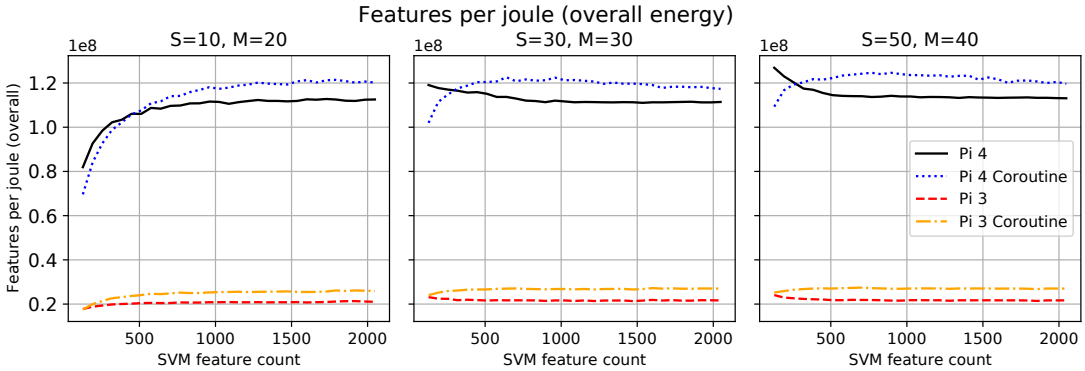
$$\text{Features per joule (overall energy)} = \frac{S * M * F}{OE} \tag{5.13}$$

$$\text{Features per joule (task energy)} = \frac{S * M * F}{TE} \tag{5.14}$$

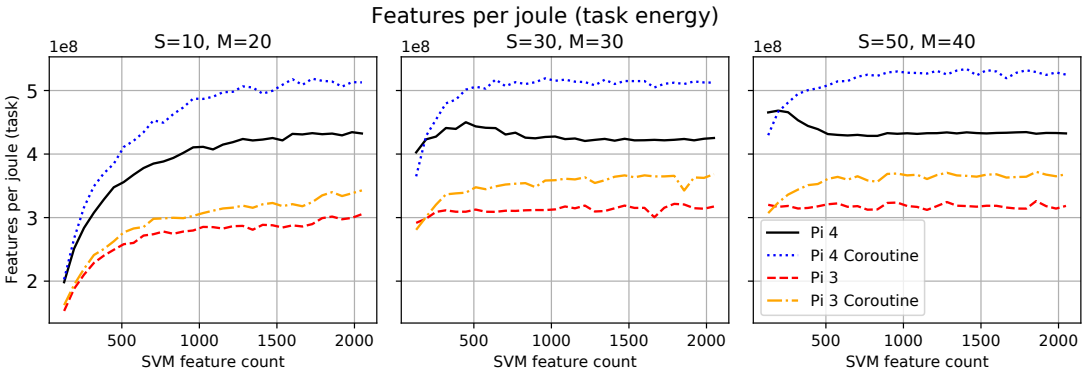
Fig. 5.13 displays these base performance statistics for both execution patterns on both platforms, across a range of sensor (S) and measurement (M) counts against SVM



(a) The effect of SVM feature count on speed of processing, measured in features per second.



(b) The effect of SVM feature count on overall energy used in processing, measured in features per joule.



(c) The effect of SVM feature count on task energy used in processing, measured in features per joule.

Figure 5.13: The effect of SVM feature count on base performance on each platform in terms of speed, overall energy and task energy. Each point represents the median value for a single experiment repeated 30 times, with outliers removed as described in Section 5.3.7. *S*: Number of sensors; *M*: Number of measurements per sensor.

feature count (F). The consistency of pattern is notable: all performance statistics tend towards a flat response (with regard to feature count) as the count increases beyond 1000.

5.4.6.1 Time

The two platforms have very different performance capabilities. The speed characteristics in Fig. 5.13a show that the Raspberry Pi 4 B performs the SVM analysis about 6 times as fast as the Raspberry Pi 3 B+.

5.4.6.2 Overall energy

Fig. 5.13b shows that the overall energy cost for the newer platform is about 5.5 times lower than that of the older platform. This ratio is consistent across sample and measurement counts.

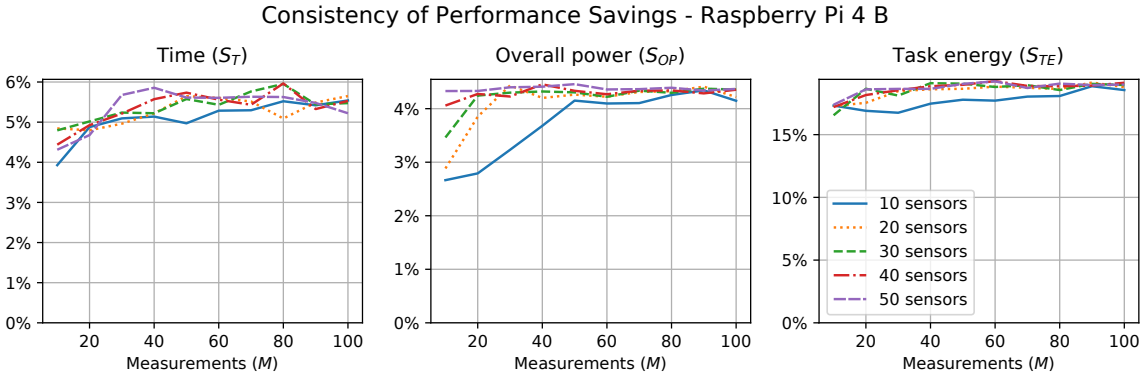
5.4.6.3 Task energy

We see from Fig. 5.13c that the difference between the two platforms is much **less** pronounced for the task energy consumption – i.e. after the cost of running the operating systems and other background processes is removed. Enhancements to the design of the Raspberry Pi 4 have resulted in general improvements to the energy usage of the platform, as can be observed in the comparative overall energy costs. The relatively smaller improvements in task energy usage for this task encourages the conclusion that these enhancements do not apply equally strongly to the types of work performed by this task, which is composed primarily of memory- and CPU-intensive operations.

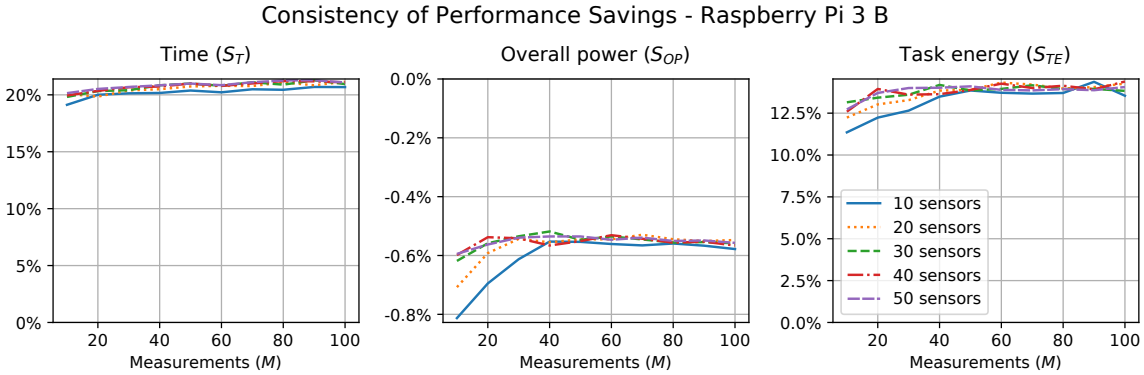
5.4.6.4 Impact of coroutine execution model

Comparing the impact of the coroutine execution model on the two platforms – as shown in Fig. 5.13 – we see improvements in all three performance metrics: execution time, overall energy consumption and task energy consumption. In all three cases – outside of the very low feature counts – there is a clear improvement in performance on both platforms, and the performance enhancement is visible across a wide range of sensor counts, measurement counts and feature counts.

5.4.7 Consistency of results



(a) Summary of best performance savings on Raspberry Pi 4 B.



(b) Summary of best performance savings on Raspberry Pi 3 B.

Figure 5.14: Summary of best performance savings on each tested platform. Each chart shows the best performance savings for each metric: S_T, S_{OP} and S_{TE} for the specified sensor and measurement count. These savings are as defined in Eqs. (5.2), (5.6) and (5.9).

Fig. 5.14a shows the consistency of the savings achievable in execution time, overall power consumption and task energy consumption across the various combinations of sensor counts and measurements per sensor, for the Raspberry Pi 4. For each metric the savings were reasonably consistent, with the exception of the cases which had the smallest number of sensors and measurements per sensor. We observe that in general savings for all three metrics rise as the measurement count per sensor is increased.

Fig. 5.14b, which summarises the same savings on the Raspberry Pi 3 B, paints a very different picture, but again a consistent one: execution time savings are 4x higher than on the Raspberry Pi 4 B, overall power consumption savings are small and negative (between -0.5% and -0.65%) and the task energy consumption savings are large (between 10% and 13%) due to the improved speed.

5.5 Discussion

5.5.1 Overall performance savings

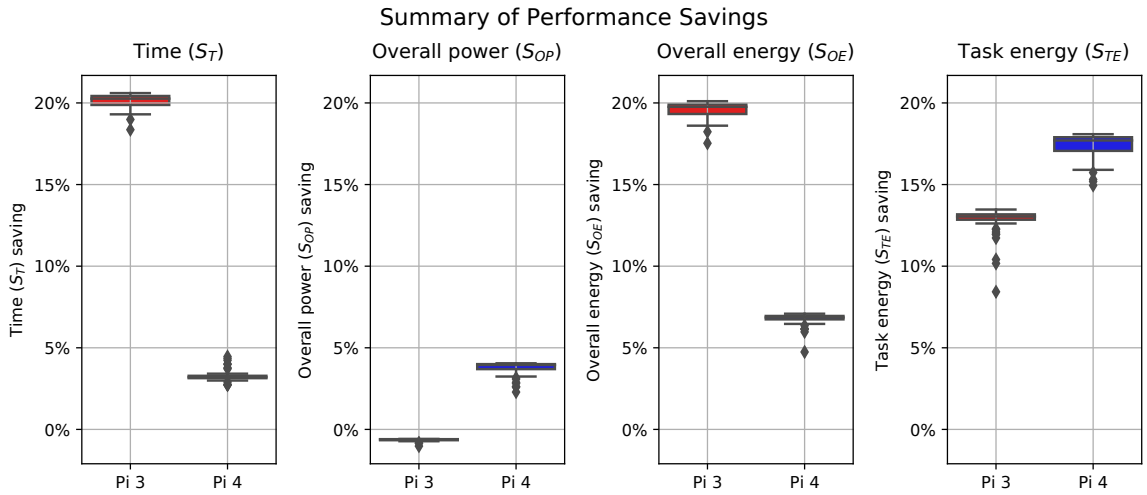


Figure 5.15: Summary of best performance savings in time and energy on each platform. The y values represent the mean performance saving for each criterion (S_T, S_{OP}, S_{OE} and S_{TE}) for each specific sensor and measurement count. These savings are as defined in Eqs. (5.2), (5.6), (5.4), and (5.9). For each combination of sensor count (S) and measurement count (M), the mean performance gain across SVM feature counts $>$ 1024 is collated. This set is shown in the box plot: the box contains the quartiles and the whiskers extend to show the rest of the range, with the exception of outliers, which are shown as dots. (The negative value for S_{OP} on the Pi 3 indicates that the modified algorithm had a net cost - more power was required by the modified coroutine execution pattern than by the unmodified sequential execution pattern.)

Performance gains are summarised in Fig. 5.15, which compares the range of savings achieved for time (S_T), overall power (S_{OP}), overall energy (S_{OE}) and task energy (S_{TE}) on the two test platforms. Time savings on both platforms are positive and show little variation across different sensor and measurement counts. The time savings on the Pi 3 platform are very large, at around 20.5%, and the time savings for the Pi 4 are lower but still useful at around 3.5%.

The overall power savings on the Pi 4 have a mean of $\approx$ 3.9% and are asymptotic to 4%; the bulk of results are over 3.2% and even the outliers remain above 2.2%. The overall power usage on the Pi 3 platform is made worse through the application of coroutines: there is a net loss of performance of $\approx$ 0.5%.

The energy consumed specifically by the SVM calculation task shows important improvements on both platforms: on the Pi 4 the energy use is improved by a mean of 19% and is asymptotic to 18%; there are outliers as low as 15%, which still represents a

valuable gain. On the Pi 3 the improvement, with a mean and an asymptote of 13%, is less marked but is still appreciable.

In Fig. 5.13, we can see a consistent pattern for the impact of SVM feature count on time, overall energy and task energy. In general, savings do not begin until the feature count reaches about 256: this is the point at which the cost – in time and energy – of creating, invoking and managing a coroutine is outweighed by the performance savings attributable to improved memory access patterns. The level of savings develops from 512 to 1024 features and stays fairly static thereafter.

The pattern differs for the smallest data size shown (10 sensors with 20 measurements each): the savings in time and overall energy do not present until the feature count reaches 512, and savings do not stabilise until around 1800. This indicates that – for this smaller number of vectors – the CPU memory cache is not filled until the vectors are proportionately larger.

5.5.2 Comparison of platforms

While the Raspberry Pi 4 B is capable of performance much superior to its Pi 3 predecessor with regard to both speed of execution of the SVM calculations (Fig. 5.13a) and overall power used (Fig. 5.13b), we can observe in Fig. 5.13c that both platforms use similar amounts of energy specifically for the SVM process. We can also see in Fig. 5.13c, by comparing the features per joule for the coroutine execution model and the sequential model, that the power savings achieved by applying the coroutine execution model are similar and very clear, tending towards 13% and 18% on the Pi 3 and 4 respectively.

Table 5.4: Summary of performance savings

Platform	Time (S_T)	Overall power (S_{OP})	Overall energy (S_{OE})	Task energy (S_{TE})
Raspberry Pi 3 B+	20.5%	-0.5%	20.0%	13.0%
Raspberry Pi 4 B	3.5%	4.0%	7.5%	18.0%

1. S_T, S_{OP}, S_{OE} & S_{TE} are calculated as defined in Eqs. (5.2), (5.6), (5.4) and (5.9) respectively.
2. For each sensor x measurements (S x M) pair, the SVM feature count with the best median outcome is selected.
3. Values of S x $M < 1024$ are excluded because the curve only becomes asymptotic above this value.
4. For each statistic the asymptotic value as S x M increases is shown.

5.5.3 Performance trade-offs

The study compares the performance characteristics of two execution models applied to an iterative calculation task: sequential execution and a switching approach using coroutines to swap cheaply between sub-regions of the task. We explored a substantial test space, varying the sizes of the outer iterations and the amount of memory used, and we observed a variety of performance changes – both positive and negative – across the space explored; we also determined that a large region within the test space reliably offered useful performance improvements.

However, it is important to note that all these tests relied on the use of a simple and unconditional code injection strategy as illustrated in Fig. 5.16a: the frequency of the task-switching was fixed. However, if a more flexible code injection strategy was used, such as that shown in Fig. 5.16b – where a test is applied, so that the task switches only when no more precached memory is available – then the cost of executing the injected code becomes unacceptably high. Using this strategy, we were unable to achieve any reliable performance improvements at all, across the same parameter space.

```
1 // Apply SVM to each row - suspend each vector
2 for (sample = 0; sample < sample_count; sample++,
      x += row_len, result_ptr++)
3 {
4   x = prefetcher.prefetch(x, data_line_count);
5   co_await std::suspend_always{};
6   *result_ptr = svm_infer(w, x, bias, row_len);
7 }
```

(a) Modification to suspend unconditionally

```
1 // Apply SVM to each row - suspend on demand
2 for (sample = 0; sample < sample_count; sample++,
      x += row_len, result_ptr++)
3 {
4   if (x + row_len > x_pre + data_loaded) {
5     x_pre = prefetcher.prefetch(x, data_line_count);
6     co_await std::suspend_always{};
7   }
8   *result_ptr = svm_infer(w, x, bias, row_len);
9 }
```

(b) Modification to suspend on exhausted cache

Figure 5.16: Modifications to the SVM application code. The added code is shown boxed. In (a) line 4 initiates a memory prefetch for the next input vector; line 5 yields to the scheduler; when other tasks have yielded, control returns to line 6, by which time vector `x` will have been loaded into CPU cache. In (b) the modification uses a different injection strategy, so that the code only suspends on-demand i.e. when the available prefetched memory is insufficient for the current operation. This strategy was found, experimentally, to be too expensive: the cost of the test outweighed any benefit from memory access pattern improvements.

In summary, there is evidence that performance in speed and energy use can be improved by a low-cost task-switching strategy that spreads memory access more evenly across the memory address space of the CPUs studied, but the mechanism that implements the strategy must be carefully managed with regard to its speed and frequency of invocation.

5.5.4 Dimensionality

As stated in Section 5.4.1.2 above, many of the results examined here, including those in Figs. 5.8, 5.9, 5.10 and 5.13, measure the impact of SVM feature count. Because feature count is driven by sensor hardware, it is a parameter that cannot be controlled by the application developer (or at run-time).

The method studied in this chapter suits problems where at least some of the dimensionality parameters are controllable; if there is no freedom to tune such parameters, then the technique should not be used.

In other edge computing applications, whether within ML or outside, the efficacy of the execution strategy described here will depend on other parameters – similar to SVM feature count – whose value is fixed by the implementation. It is important to test across the range of likely values for an application instance before investing in the strategy.

5.5.5 Value of mini-scheduler

We have observed in Section 5.5.3 that the implementation of the injected code that contains the machinery for task suspension can be the deciding factor in whether the strategy is successful or not; similarly, the implementation of the mini-scheduler will have an important impact on the efficacy of the strategy.

The simple round-robin pattern summarised in Algorithm 2 (and listed in full in Section 5.7.4) has low overheads but does not have enough per-coroutine state information to usefully prioritise between coroutines. It is possible that a more complex prioritisation mechanism would offer further performance benefits; alternatively, a trade-off between complexity and performance might result in the opposite outcome.

5.6 Conclusions

We have investigated the use of an algorithmic transformation of C++ code to improve runtime performance on an edge device. This transformation uses coroutines and a "mini-scheduler" class to improve the performance of multi-layered highly iterative code – the type of code typically found in machine learning inference applications.

We conducted experiments on two edge gateway devices: a Raspberry Pi 4 B and a Raspberry Pi 3 B+. We measured the impact of the coroutine execution model on

the performance characteristics of a support vector machine on a gateway, which locally processed feature vectors passed in by multiple sensors. We varied the number of sensor nodes connected to the gateway device, the number of measurements made by each sensor, and the size of the feature vectors. Table 5.4 shows a summary of the results.

We observed clear improvements in speed of performance: 3.5% on the Pi 4 and 20.5% on the Pi 3. Notably, we also observed - on the Pi 4 only - a substantive reduction in the overall power used by the system while calculating the SVM: 4.0%.

Combining the impact of time savings and overall power savings, we observed an overall energy saving of 7.5% on the Pi 4 and 20% on the Pi 3. (The energy saving on the Pi 3 is solely a side-effect of the time saving).

Separating out the power used specifically by the SVM calculation, we saw a reduction of 18.0% on the Pi 4 and 13% on the Pi 3.

We observed that the technique offers useful benefits on both of the platforms studied, but in differing ways, as summarised in Table 5.4: the Pi 4 offers savings in the 3.5% to 4.0% region for both time and overall power, whereas the Pi 3 offered time savings up to 20.5% and a slight power cost.

This study was restricted to testing on Raspberry Pi devices. Our earlier work [16] established that Intel platforms could also show speed benefits; it would be useful to investigate whether Intel platforms – particularly the small devices used for edge processing – will also display power usage improvements.

The use of this transformation on existing application code offers considerable cost savings. The throughput improvements observed would allow a proportional reduction in the number of gateways in a WSN, with a positive impact on equipment and deployment costs. The energy improvements – up to $\approx$ 18% – offer large increases in battery life, with consequent deployment and maintenance savings.

The benefits discussed here are part of a trade-off that places increased code complexity against reduced execution time, energy use and peak current. We assert that - with the use of C++20 - the increase in code complexity is known and manageable: the scheduler code in Listing 5.1 is easily and immediately applicable to other problem domains; the alterations to the existing iteration code in Listing 5.2 are small and simple; and the invocation of the iterator/scheduler, as shown in Listing 5.3, is transparent and short.

5.7 Appendices

5.7.1 Detailed results

Tables 5.5 and 5.6 contain a summary of the results for each sensor count/measurement count combination for the Raspberry Pi 4B and 3B+ platforms respectively. All statistics are percentages; they represent the change in performance (i.e. the reduction in cost) achieved through applying the coroutine-based execution pattern described in Section 5.3. The formulae for the measures are presented in Eqs. (5.2), (5.6), (5.9) and (5.11).

Table 5.5: Raspberry Pi 4 B: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
10	10	3.93	2.35	1.26	2.66	2.10	0.45	17.32	14.91	1.51	2.78	2.38	0.25
10	20	4.88	3.99	0.81	2.79	2.53	0.20	16.90	15.63	0.79	2.76	2.42	0.17
10	30	5.09	4.30	0.57	3.23	2.72	0.29	16.74	15.87	0.73	3.21	2.59	0.26
10	40	5.14	4.21	0.75	3.68	2.95	0.41	17.47	16.07	0.87	3.51	2.77	0.43
10	50	4.97	3.95	0.90	4.15	3.17	0.60	17.78	16.45	0.86	4.15	3.05	0.58
10	60	5.29	3.72	1.34	4.10	3.34	0.58	17.71	16.52	0.89	3.77	3.10	0.53
10	70	5.30	3.65	1.49	4.10	3.47	0.54	18.03	16.71	0.99	4.02	3.27	0.45
10	80	5.52	3.64	1.48	4.26	3.59	0.53	18.08	17.06	0.78	4.05	3.36	0.47
10	90	5.42	3.76	1.37	4.34	3.78	0.48	18.86	17.51	1.00	4.75	3.61	0.69
10	100	5.54	3.74	1.56	4.15	3.71	0.41	18.55	17.32	0.82	3.98	3.46	0.44
20	10	4.84	3.69	0.78	2.88	2.49	0.26	17.28	15.04	1.08	2.86	2.42	0.19
20	20	4.80	4.03	0.58	3.85	2.99	0.45	17.54	15.97	0.98	3.43	2.85	0.37
20	30	4.96	3.67	1.20	4.43	3.41	0.69	18.46	16.74	1.18	4.75	3.29	0.67
20	40	5.22	3.67	1.33	4.20	3.62	0.53	18.61	17.12	0.90	4.02	3.30	0.51
20	50	5.65	3.64	1.52	4.26	3.71	0.44	18.65	17.26	0.74	4.16	3.32	0.43
20	60	5.60	3.68	1.47	4.22	3.78	0.37	18.83	17.68	0.72	4.37	3.42	0.59
20	70	5.52	3.54	1.44	4.31	3.84	0.33	18.74	17.60	0.68	4.21	3.20	0.61
20	80	5.09	3.63	1.27	4.33	3.84	0.31	18.64	17.75	0.60	3.95	3.13	0.54
20	90	5.49	3.68	1.36	4.41	3.83	0.28	19.16	17.75	0.63	3.92	3.13	0.42
20	100	5.65	3.61	1.50	4.22	3.84	0.25	18.78	17.72	0.63	4.06	3.05	0.45

Table 5.5: Raspberry Pi 4 B: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
30	10	4.80	3.61	0.62	3.46	2.75	0.36	16.54	14.96	1.05	3.29	2.71	0.37
30	20	5.02	3.46	1.21	4.24	3.41	0.59	18.66	16.49	1.12	4.36	3.30	0.53
30	30	5.24	3.59	1.36	4.30	3.68	0.46	18.12	17.14	0.89	4.19	3.46	0.61
30	40	5.22	3.54	1.50	4.32	3.77	0.33	19.10	17.51	1.01	4.26	3.42	0.53
30	50	5.58	3.60	1.46	4.30	3.83	0.30	19.07	17.75	0.72	4.31	3.02	0.63
30	60	5.43	3.71	1.38	4.22	3.87	0.28	18.80	17.94	0.57	3.65	3.08	0.41
30	70	5.76	3.65	1.49	4.34	3.87	0.29	18.94	17.94	0.62	4.00	3.14	0.42
30	80	5.95	3.59	1.39	4.30	3.86	0.25	18.55	17.82	0.77	3.79	3.07	0.37
30	90	5.44	3.69	1.31	4.37	3.95	0.26	19.05	18.02	0.66	3.78	2.98	0.44
30	100	5.47	3.67	1.34	4.36	3.89	0.28	18.96	17.88	0.71	3.66	2.90	0.40
40	10	4.44	3.38	0.76	4.06	3.04	0.52	17.20	15.46	1.09	4.06	2.95	0.53
40	20	4.92	3.16	1.28	4.27	3.64	0.55	18.16	16.75	1.04	4.20	3.44	0.55
40	30	5.21	3.54	1.29	4.23	3.81	0.35	18.54	17.58	0.61	3.86	3.37	0.42
40	40	5.57	3.47	1.44	4.45	3.87	0.36	18.86	17.68	0.81	4.18	3.19	0.54
40	50	5.73	3.62	1.46	4.33	3.86	0.30	18.97	17.76	0.82	4.36	3.18	0.57
40	60	5.56	3.60	1.49	4.27	3.90	0.27	19.27	17.87	0.81	3.89	3.11	0.47
40	70	5.43	3.63	1.34	4.34	3.92	0.29	18.88	18.01	0.61	4.00	3.02	0.50
40	80	5.96	3.64	1.46	4.34	3.89	0.25	18.86	17.90	0.84	3.88	3.00	0.35
40	90	5.33	3.68	1.32	4.29	3.93	0.26	18.91	18.10	0.63	3.61	2.95	0.40
40	100	5.53	3.60	1.44	4.35	3.90	0.25	19.14	17.97	0.87	3.87	2.90	0.47
50	10	4.31	2.94	0.98	4.33	3.34	0.60	17.38	15.90	1.20	4.45	3.25	0.60
50	20	4.68	3.13	1.35	4.33	3.83	0.45	18.61	17.22	0.80	4.62	3.57	0.49
50	30	5.68	3.51	1.31	4.40	3.88	0.31	18.65	17.65	0.56	4.05	3.44	0.41
50	40	5.86	3.63	1.41	4.41	3.93	0.30	18.65	17.95	0.59	4.11	3.18	0.37
50	50	5.61	3.64	1.41	4.46	3.90	0.31	19.05	17.87	0.66	4.05	3.19	0.55
50	60	5.61	3.67	1.44	4.36	3.93	0.29	19.25	18.03	0.72	4.10	2.96	0.43
50	70	5.63	3.59	1.29	4.36	3.93	0.29	18.72	18.03	0.62	3.93	2.95	0.49
50	80	5.63	3.67	1.39	4.39	3.91	0.27	19.07	18.01	0.70	3.99	2.91	0.50

Table 5.5: Raspberry Pi 4 B: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
50	90	5.48	3.69	1.37	4.34	3.92	0.25	18.95	18.04	0.65	5.63	3.08	0.73
50	100	5.22	3.64	1.39	4.37	3.91	0.27	18.99	18.01	0.82	3.94	3.10	0.53

Notes: (i) S=Samples; M=Measurements. (ii) All statistics are percentages. Each represents the change in a cost (time, energy or power) achieved as a result of switching from a standard sequential execution pattern to an interleaved coroutine-based execution pattern. Positive numbers represent improvements. (iii) S_T is defined in Eq. (5.2). (iv) S_{OP} is defined in Eq. (5.6). (v) S_{TE} is defined in Eq. (5.9). (vi) S_{PP} is defined in Eq. (5.11).

Table 5.6: Raspberry Pi 3 B+: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
10	10	19.12	18.02	0.79	-0.81	-1.05	0.09	11.35	8.13	1.60	1.19	-0.85	0.56
10	20	20.01	19.07	0.57	-0.69	-0.89	0.13	12.23	9.67	1.78	1.11	-0.56	0.73
10	30	20.13	19.42	0.41	-0.61	-0.80	0.17	12.65	11.17	1.21	1.46	-0.43	0.74
10	40	20.17	19.57	0.32	-0.55	-0.77	0.21	13.49	11.81	1.17	0.32	-0.62	0.47
10	50	20.38	19.80	0.32	-0.55	-0.70	0.14	13.85	12.36	1.16	0.41	-0.46	0.49
10	60	20.22	19.86	0.20	-0.56	-0.71	0.14	13.72	12.66	0.73	0.18	-0.38	0.33
10	70	20.49	20.07	0.26	-0.57	-0.68	0.09	13.67	12.63	0.62	-0.03	-0.52	0.34
10	80	20.44	19.94	0.28	-0.56	-0.66	0.07	13.70	12.73	0.64	1.04	-0.33	0.52
10	90	20.69	20.11	0.34	-0.57	-0.67	0.08	14.37	12.89	0.62	0.57	-0.25	0.49
10	100	20.68	20.09	0.26	-0.58	-0.67	0.12	13.54	12.74	0.42	0.40	-0.50	0.44
20	10	20.08	18.81	0.60	-0.71	-0.90	0.13	12.24	9.37	1.93	2.01	-0.94	1.27
20	20	19.85	19.55	0.21	-0.59	-0.74	0.14	13.02	11.70	1.21	0.61	-0.39	0.46
20	30	20.39	19.85	0.27	-0.54	-0.70	0.10	13.28	12.38	0.78	0.03	-0.60	0.59
20	40	20.50	19.87	0.45	-0.56	-0.68	0.13	13.85	12.66	0.67	0.12	-0.46	0.33
20	50	20.74	20.03	0.38	-0.55	-0.68	0.11	13.90	12.78	0.63	0.27	-0.49	0.35
20	60	20.79	20.25	0.27	-0.55	-0.61	0.05	14.33	13.02	0.70	0.79	-0.43	0.40

Table 5.6: Raspberry Pi 3 B+: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
20	70	20.77	20.33	0.23	-0.53	-0.62	0.06	14.21	12.98	0.69	0.61	-0.41	0.48
20	80	21.09	20.35	0.26	-0.54	-0.61	0.05	13.97	13.07	0.73	0.81	-0.48	0.51
20	90	20.78	20.39	0.18	-0.55	-0.62	0.04	14.06	12.95	0.47	0.98	-0.38	0.56
20	100	21.15	20.40	0.25	-0.55	-0.62	0.06	14.26	12.91	0.69	0.39	-0.56	0.44
30	10	19.82	19.35	0.23	-0.62	-0.75	0.13	13.15	11.41	1.39	1.72	-0.09	0.88
30	20	20.33	19.82	0.20	-0.56	-0.66	0.09	13.42	12.29	0.82	1.82	-0.38	0.58
30	30	20.41	20.05	0.21	-0.53	-0.64	0.09	13.60	12.67	0.65	0.77	-0.53	0.49
30	40	20.84	20.19	0.21	-0.52	-0.61	0.07	14.19	13.05	0.80	1.26	-0.36	0.59
30	50	21.00	20.22	0.27	-0.55	-0.60	0.05	13.86	12.99	0.58	1.33	-0.39	0.54
30	60	20.66	20.33	0.16	-0.54	-0.60	0.06	13.95	13.01	0.58	0.59	-0.46	0.55
30	70	21.10	20.42	0.26	-0.55	-0.60	0.05	14.16	13.10	0.42	0.29	-0.53	0.38
30	80	20.90	20.43	0.21	-0.56	-0.61	0.04	13.94	13.03	0.48	0.59	-0.75	0.49
30	90	21.28	20.49	0.27	-0.55	-0.62	0.05	13.95	12.96	0.69	0.52	-0.67	0.33
30	100	20.95	20.47	0.24	-0.56	-0.62	0.05	13.83	12.94	0.50	0.12	-0.69	0.33
40	10	19.94	19.39	0.30	-0.60	-0.72	0.10	12.59	11.55	0.93	0.91	-0.31	0.89
40	20	20.35	19.91	0.23	-0.54	-0.65	0.09	13.95	12.56	0.88	0.96	-0.41	0.51
40	30	20.67	20.22	0.27	-0.54	-0.63	0.05	13.61	12.79	0.49	0.31	-0.60	0.43
40	40	20.73	20.30	0.23	-0.57	-0.62	0.05	13.63	12.97	0.45	0.67	-0.48	0.45
40	50	20.99	20.34	0.22	-0.55	-0.62	0.05	13.88	12.96	0.57	0.73	-0.59	0.55
40	60	20.79	20.37	0.16	-0.53	-0.60	0.05	14.26	13.28	0.67	0.84	-0.59	0.47
40	70	21.00	20.50	0.23	-0.54	-0.62	0.05	14.00	13.19	0.47	0.58	-0.55	0.43
40	80	21.21	20.54	0.25	-0.56	-0.63	0.06	14.14	13.09	0.67	-0.39	-0.77	0.15
40	90	21.08	20.53	0.22	-0.55	-0.63	0.05	13.92	12.97	0.58	-0.32	-0.75	0.23
40	100	21.00	20.48	0.21	-0.57	-0.63	0.05	14.39	13.06	0.62	4.55	-0.56	1.15
50	10	20.15	19.42	0.30	-0.59	-0.69	0.10	12.72	11.86	0.84	1.33	-0.52	0.57
50	20	20.52	20.00	0.21	-0.56	-0.62	0.06	13.71	12.84	0.72	0.62	-0.35	0.53
50	30	20.69	20.27	0.22	-0.54	-0.62	0.05	14.00	12.76	0.59	0.57	-0.51	0.47
50	40	20.84	20.31	0.19	-0.54	-0.61	0.05	14.02	13.08	0.62	0.56	-0.62	0.37

Table 5.6: Raspberry Pi 3 B+: Summary of results for each sensor/measurement count

S	M	Time S_T			Overall power S_{OP}			Task energy S_{TE}			Peak power S_{PP}		
		Max	Mean	sd	Max	Mean	sd	Max	Mean	sd	Max	Mean	sd
50	50	20.99	20.39	0.22	-0.54	-0.61	0.06	14.10	12.90	0.73	0.49	-0.65	0.33
50	60	20.85	20.44	0.22	-0.55	-0.61	0.05	13.89	13.05	0.48	0.67	-0.76	0.55
50	70	21.11	20.45	0.24	-0.54	-0.61	0.05	13.86	13.18	0.54	0.16	-0.75	0.38
50	80	21.26	20.53	0.27	-0.55	-0.62	0.06	13.93	12.95	0.62	1.17	-0.69	0.47
50	90	21.29	20.49	0.24	-0.55	-0.61	0.05	13.87	13.10	0.49	0.40	-0.61	0.55
50	100	21.09	20.58	0.24	-0.56	-0.63	0.05	14.06	12.99	0.51	-0.14	-0.75	0.24

Notes: (i) S=Samples; M=Measurements. (ii) All statistics are percentages. Each represents the change in a cost (time, energy or power) achieved as a result of switching from a standard sequential execution pattern to an interleaved coroutine-based execution pattern. Positive numbers represent improvements. (iii) S_T is defined in Eq. (5.2). (iv) S_{OP} is defined in Eq. (5.6). (v) S_{TE} is defined in Eq. (5.9). (vi) S_{PP} is defined in Eq. (5.11).

5.7.2 Platform hardware characteristics

Table 5.7: Test platform characteristics

Name	Pi 4	Pi 3
Computer	Raspberry Pi 4 B	Raspberry Pi 3 B+ [181]
Released	2019	2018
CPU	ARM Cortex A72 64-bit SOC	ARM Cortex A53 64-bit SOC
Memory	2GB RAM	1GB RAM
Cores	Quad-core @ 1.5 GHz	Quad-core @ 1.4 GHz
CPU cache	32 KiB L1 + 1 MiB L2 Cache	16 KiB L1 + 512 KiB L2 Cache

5.7.3 Algorithms

Algorithm 2 Operation of mini-scheduler for coroutines

```
1:  procedure RUN(cc, n, coro)          ▷ Run coroutine 'coro' for each of 'n' items,
2:                                      ▷ using 'cc' coroutines.
3:      coros ← [1, ..., cc]            ▷ Create an array of coroutines
4:      done ← [1, ..., cc]             ▷ Array of completion flags for each coroutine
5:      for i = 1, ..., cc do
6:          coros[i] ← new coroutine(coro, i)   ▷ Create new coroutine to process item i
7:          done[i] ← false             ▷ Coroutine's initial state is 'incomplete'
8:      next ← cc + 1                   ▷ Index for next item to be processed
9:      completed ← 0
10:     while completed < n do          ▷ Continue until all work items are completed
11:         for i = 1, ..., cc do       ▷ Inspect each coroutine in turn – round-robin
12:             if not done[i] then     ▷ If any work remains ...
13:                 if not coros[i].is_complete() then
14:                     coros[i].resume()   ▷ If not complete – continue
15:                 else
16:                     if next == n then   ▷ Test whether any work items remain
17:                         done[i] ← true  ▷ No work remains – mark this slot as complete
18:                     else
19:                         coros[i] ← new coroutine(coro, next)  ▷ Replace coroutine with a
20:                                         ▷ new instance
21:                         next ← next + 1
22:                     completed ← completed + 1
```

5.7.4 Source code

Listing 5.1 contains the mini-scheduler used in these experiments, a generalised C++ template class. Listing 5.2 shows the SVM iteration algorithm to which the scheduler was applied, both before and after the changes that were required by the scheduler. Listing 5.3 demonstrates the invocation of the scheduler for the batch of SVM tasks.

```
1 // Generalised runner for parallelising iterative
```

```
// tasks across multiple coroutines.
template <typename REFDATA_T>
class coroutine_runner {
 public:
  typedef resumable_t (*coro_fn_t)(
    const prefetcher_t &prefetcher,
    REFDATA_T &refdata, size_t coroutine_index);

  coroutine_runner(const prefetcher_t &prefetcher,
    REFDATA_T &refdata)
    : prefetcher_(prefetcher), refdata_(refdata) {}

  void run(size_t coroutine_count, size_t item_count,
           coro_fn_t coro_fn) {
    // A collection of parallelised coroutines
    std::vector<resumable_t> tasks;
    // Status of all items
    std::vector<bool> done(coroutine_count, false);
    size_t incomplete = item_count;

    // Create coroutines, ready to run. There will always be at most
    // coroutine_count of them, and each will be deleted and replaced
    // by a new one when its item is complete.
    for (size_t b = 0; b < coroutine_count; b++) {
      tasks.push_back(coro_fn(prefetcher_, refdata_, b));
    }

    // Work through all tasks until all are done
    size_t next_item = coroutine_count;
    while (incomplete > 0) {
      for (size_t c = 0; c < tasks.size(); c++) {
        if (done[c]) {
          continue;
        }
        resumable_t &t = tasks[c];
        if (t.is_complete()) {
          if (next_item < item_count) {
            tasks[c] = coro_fn(prefetcher_, refdata_,
              next_item);
            next_item++;
          } else {
            done[c] = true;
          }
          incomplete--;
        } else {
          t.resume();
        }
      }
    }
  }
```

```
52
53  protected:
54   const prefetcher_t &prefetcher_;
55   REFDATA_T &refdata_;
56 };
```

Listing 5.1: Generalised C++ template to apply the coroutined execution context to any function after it has been converted to a coroutine with three parameters: (i) prefetcher class, (ii) an application-dependent context class and (iii) an index into the collection of sub-tasks (as defined by `typedef coro_fn_t` in lines 5-7).

```
1  // SVM iteration algorithm, before being refactored as a coroutine
2  void infer_sensor_sequential(runtime_data &rt_data, uint32_t
       sensor_index)
3  {
4    const data_item_t *x, *w;
5
6    // Resolve weights & bias for this sensor
7    w = rt_data.resolve_w(sensor_index);
8
9    // Inspect w data
10   data_item_t bias = w[0];
11   w++;
12
13   // Get sensor data base
14   const data_vector_t& x_vec = rt_data.resolve_x_vec(sensor_index);
15   x = x_vec.data();
16   auto row_len = rt_data.rt.sv_len;
17   auto sample_count = x_vec.size() / row_len;
18
19   // Get result base
20   std::vector<result_t>& results = rt_data.resolve_results_vec(
21       sensor_index);
22   result_t* result_ptr = results.data();
23
24   for (uint32_t sample = 0; sample < sample_count; sample++, x +=
       row_len,
25        result_ptr++)
26   {
27     *result_ptr = svm_infer(w, x, bias, row_len) ? 1 : 0;
28   }
29 }
30
31 // Controller to run task sequentially
32 void run_infer_sequential(runtime_data &rt_data)
33 {
34   for (uint32_t i = 0; i < rt_data.rt.sensor_count; i++)
35   {
36     infer_sensor_sequential(rt_data, i);
37   }
```

```
}

// SVM iteration algorithm, after refactoring as a coroutine
static resumable infer_sensor_coro(const prefetcher_t &prefetcher,
    runtime_data &rt_data, size_t coroutine_index)
{
  uint32_t sensor_index = (uint32_t)coroutine_index;
  co_await CORO_STD::suspend_always{};

  const data_item_t *x, *w;

  // Calculate prefetch sizes
  size_t weights_size = rt_data.rt.w_len * sizeof(data_item_t);
  size_t weights_line_count = to_pf_line_count(weights_size);

  // Resolve weights & bias for this sensor
  w = rt_data.resolve_w(sensor_index);
  w_next = prefetcher.prefetch(reinterpret_cast<const char*>(w),
      weights_line_count);
  co_await CORO_STD::suspend_always{};

  // Inspect w data
  data_item_t bias = w[0];
  w++;

  // Get sensor data base
  const data_vector_t& x_vec = rt_data.resolve_x_vec(sensor_index);
  x = x_vec.data();
  auto row_len = rt_data.rt.sv_len;
  auto sample_count = x_vec.size() / row_len;

  // Calculate prefetch sizes
  size_t data_size = row_len * sizeof(data_item_t);
  size_t data_line_count = to_pf_line_count(data_size);
  size_t results_size = sample_count * sizeof(result_t);
  size_t results_line_count = to_pf_line_count(results_size);

  // Get result base
  std::vector<result_t>& results = rt_data.resolve_results_vec(
      sensor_index);
  result_t* result_ptr = results.data();
  result_next = prefetcher.prefetchw(reinterpret_cast<char*>(result_ptr
    ),
      results_line_count);

  for (uint32_t sample = 0; sample < sample_count; sample++, x +=
    row_len,
      result_ptr++)
  {
    x_next = prefetcher.prefetch(reinterpret_cast<const char*>(x),
```

```
86            data_line_count);
87      co_await CORO_STD::suspend_always{};
88      *result_ptr = svm_infer(w, x, bias, row_len) ? 1 : 0;
89    }
90  }
```

Listing 5.2: Sample task - `infer_sensor_sequential()` - to calculate SVM for each of a set of $n_s \, x \, n_p$ vectors. The task is reorganised as a coroutine – `infer_sensor_coro()` – ready to be run by the mini-scheduler, `coroutine_runner::run()`.

```
1  // Declare instance of platform-dependent memory prefetcher
2  prefetcher_t prefetcher;
3
4  // Declare an instance of the execution context template,
5  // which references the application-dependent runtime data
6  // rt_data.
7  coroutine_runner<runtime_data> runner_with_prefetch(
8    prefetcher, rt_data);
9
10 // Invoke the execution context instance to run the
11 // coroutined function across all instances of the sensor
12 // data sets.
13 runner_with_prefetch.run(rt_data.task_count,
14   rt_data.sensor_count, infer_sensor_coro);
```

Listing 5.3: Typical usage of C++ template to run a coroutine across iterative tasks.

Chapter 6

Conclusion

Chapter Abstract

This thesis has examined the use of C++20 coroutines in software for embedded and edge computing platforms. Through the four research chapters, C++ coroutines on these platforms have been examined from a number of viewpoints. The demand for language-native coroutines in C-based development for resource-constrained devices has been examined and found to be widespread. An implementation of C++20 coroutines for bare-metal platforms has been developed, tested and critically evaluated: C++20 coroutines have been found to be effective and easy to use, but not quite ready for microcontroller use. C++ coroutines have been tested on edge devices across a variety of micro-benchmark algorithms: they were found to be an effective means by which to employ lightweight multi-threading to improve the memory access patterns of iterative code, and thereby also to improve speed of execution. Finally, C++ coroutines were used in a real-world application for edge devices and found to be straightforward to program, as well as improving not only execution speed but also power usage.

6.1 Overview

In this thesis the application of C++20 coroutines to embedded and edge computing has been examined from a variety of viewpoints. The study began in 2017, when the inclusion of coroutines in standard C++ was still being debated, and concluded in 2024, when the language feature is supported fully in all major compilers and the C++23

standard has matured the feature with the introduction of the first concrete coroutine (`std::generator`).

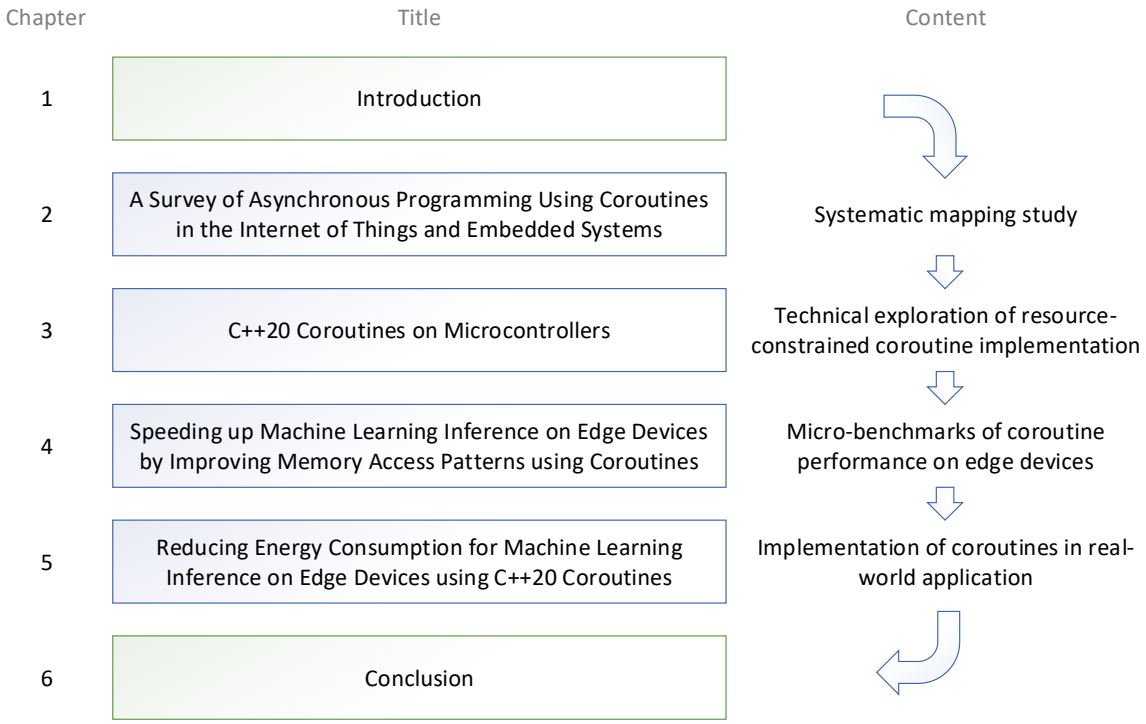


Figure 6.1: Chapter roadmap

As shown in Fig. 6.1, the thesis' research content begins in Chapter 2 with a systematic survey of all published papers that reported a need for a mechanism such as coroutines on small constrained-resource platforms. It continues in Chapter 3 with a reflection on the development of a C++20-compliant coroutine implementation for a 'bare-metal' microcontroller, and an analysis of its performance. In Chapter 4 the use of coroutines on edge devices is examined with a set of micro-benchmarks, which investigate the trade-offs in asynchronous C++ programming between the simplicity brought by coroutines and their performance costs. Finally, Chapter 5 describes and measures the use of C++ coroutines in the machine learning inference implementation module of a real-world edge computing application, converting an existing highly iterative execution pattern to a coroutine-based execution, examining coding costs and changes to execution time and energy consumption.

6.2 Summary of findings

This section summarises the findings of the four research chapters, and places them in the context of the research questions (RQs), which are repeated below:

RQ1: Can mainstream coroutine solutions apply to resource-constrained platforms?

RQ2: Are the costs deterministic?

RQ3: Can the benefits be clearly demonstrated?

6.2.1 Chapter 2

Chapter 2 (*A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems*) reviewed the literature regarding coroutine-based programming on resource-constrained platforms. Within the bounds of development for the Internet of Things (IoT) and for embedded systems, the chapter presented a systematic mapping study into the demand for a language-native solution to asynchronous code problems. On the one hand the solution should possess low programming costs and on the other it should not incur significant performance costs. The study found substantial evidence that a solution of this type was in demand.

The systematic study examined all published work that developed or used light-weight threads – whether implemented as coroutines or through other means – on microcontroller-based platforms, and collated the findings. Out of 566 candidate papers, 35 met all the selection criteria and were investigated fully. C & C++ were the programming languages used by 22 out of the 35. 16 studies used 8- and 16-bit processors, while 13 used 32-bit. The most common use cases included concurrency (17 papers), network communication (15) and sensor readings (9). The foremost intended benefit was code style and simplicity (12 papers).

A major thread of the analysis investigated some of the characteristics of the coroutine implementation: whether it managed control flow and the state of local variables. Several results of the analysis stood out. Almost all implementations managed control flow on behalf of the programmer, and our conclusion that this usability feature was a vital component of a coroutine implementation was supported by the fact that code style and simplicity led the list of intended benefits. None of the coroutines written

in native C or C++ provided managed local variable state – because the languages do not support it; but all of the non-native C implementations and almost all of the non-C papers provided this usability feature.

The study concluded that there existed widespread demand for "a formalised, stable, well-supported implementation of coroutines in C++ ... [for] resource-constrained devices, and further that such an implementation would bring benefits specific to such devices".

In the context of the research questions, Chapter 2 did not directly inform any of the three RQs. The survey found many and varied attempts to implement coroutines – or software mechanisms displaying the same behaviour as coroutines – on small platforms. One notable feature of the findings was how widely varieties of Duff's device [46] were used: given the inconvenience of using this device, and the constraints its use applies to C and C++ usage, this adds force to the survey's conclusion that there is clear evidence of a demand among C and C++ developers for coroutines on smaller platforms. Combined with the prominence of code style and simplicity as an intended benefit of the coroutine-like device, we concluded that all three RQs were worth investigating. If a mainstream (i.e. language-native) coroutine can be brought to smaller platforms, it will be found useful, so long as it satisfies the special conditions that apply on resource-constrained platforms, including deterministic behaviour (in terms both of memory usage and of performance) and avoidance of dynamic memory allocation, alongside demonstrable benefits in areas such as ease of coding, source code clarity and run-time stability.

6.2.2 Chapter 3

Chapter 3 (*C++20 Coroutines on Microcontrollers*) developed and reported on an implementation of C++20 coroutines [82] on a 'bare-metal' microcontroller (i.e. one with no operating system).

The chapter noted a number of substantive problems during the implementation, which impact on all three of our research questions.

1. **Exceptions**: The C++20 standard for coroutines specifies that the treatment of an `await-expression` must catch exceptions and rethrow them. For C++ embedded developers who are conventionally avoiding the use of exceptions through compiler flags, a special build of the `<coroutine>` header is required.

2. **Memory location**: Following the C++20 standard, it is very cumbersome to specify that the memory assigned to a coroutine's stack frame (CSF) should be stored in either the stack of the calling function or in global space.

3. **Memory usage**: Furthermore, it is not possible to tell at compile time how much memory space will be consumed by CSF instances. This can only be ascertained at link time.

As regards RQ1 (*Can mainstream coroutine solutions apply to resource-constrained platforms?*), we can conclude that in its initial state (per the C++20 standard), a mainstream coroutine solution is not ideal for smaller platforms because if its dependence on C++ exceptions, the possibility that the heap may be used for allocation of CSFs, and its unspecified run-time memory costs. Further, it was noted that the memory requirements for the coroutine infrastructure were high enough to make the feature inappropriate for extremely resource-constrained devices with very little RAM.

While the costs in terms of speed and memory usage were found to be deterministic – informing RQ2 (*Are the costs deterministic?*) – it was found that the size of those memory costs could only be quantified by adding a minor modification to the compilation tool-chain.

For RQ3 (*Can the benefits be clearly demonstrated?*), the chapter considered the trade-off between – on the one hand – the simplicity of the application code and – on the other – any performance costs in terms of speed, compiled code size and memory use. It found that the specification encouraged asynchronous application code which was easy to write and easy to understand. (Note that this conclusion did not apply to the coroutine *library implementation code*, which was not found to be straightforward to code.) Compared to a standard state-machine version, the application code contained one sixth as many source lines of code, and offered a much more intelligible continuous and direct style.

Our study found that while speed of performance was only marginally slower than the extremely efficient – but flawed – Protothreads library [49], the coroutine version was more than 12 times as fast as the thread implementations of leading real-time operating systems [15, 135] and used around half as much code memory. The chapter concluded that these clear outcomes provided a positive outcome for RQ3.

6.2.3 Chapter 4

Chapter 4 (*Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines*) examined the trade-offs involved in improving the performance of a number of micro-benchmarks performing tasks commonly delegated to edge devices, particularly within machine learning inference implementations. The benchmarks included **B+Tree**, which visited all the nodes of a B+ tree, **SVM**, a support vector machine, **Norm**, which normalised the colour values of images to the ImageNet standard [43], and **CNN**, which applied a 3x3 kernel to the pixels of a batch of images.

The study examined a large parameter space for each algorithm: three different hardware platforms were tested, and two compilers; four different numeric types were used; the number of concurrent coroutines was varied; and finally between 32 and 64 different data set sizes were used for each micro-benchmark. Each test was run 10 times, to remove outliers.

Very large performance improvements – up to 65% – were recorded for the **B+Tree** benchmark, which requires that large continuous 'chunks' of memory are retrieved from effectively random memory locations between each step of the task. This type of 'pointer chasing' task is ideal for a design that injects prefetching code, and speed benefits had already been observed (albeit on much larger and more powerful platforms) elsewhere [145, 85, 75].

The performance benefits to the **SVM** benchmark and, to a lesser extent, the **Norm** and **CNN** benchmarks demonstrated that improvements could be achieved not only through CPU cache prefetching but also through improved memory access patterns.

The changes to the benchmark in order to support the coroutine-based execution pattern were small, simple and transparent, and this informs RQ1 (*Can mainstream coroutine solutions apply to resource-constrained platforms?*). For these edge devices (smaller than conventional desktop or cloud machines, but larger than the tiny platforms used in Chapter 3), the C++20 coroutine standard and its implementation applied effectively.

The tests were applied across a very wide parameter space, and provided positive performance improvements that were strongly consistent across large areas within the parameter space. This informed both RQ2 (*Are the costs deterministic?*) and RQ3 (*Can the benefits be clearly demonstrated?*): in each case the outcome was positive.

6.2.4 Chapter 5

Chapter 5 (*Reducing Energy Consumption for Machine Learning Inference on Edge Devices using C++20 Coroutines*) looked deeper into the use of coroutines in a real C++ application, and examined the costs and benefits of coroutine use in the iterative code that typifies machine learning inference when it is located on edge devices. The outcomes built on those of Chapter 4. They confirmed the findings of Chapter 4 in the context of a full application: the use of coroutines in iterative code provided improved speed of performance and had very low programming cost. The research also discovered important improvements in the level of power used, observing savings of 7.3% on a Raspberry Pi 4 edge device. In total, we observed energy savings for the SVM calculation task of 18% on the Pi 4 and 13% on a Raspberry Pi 3.

In terms of the impact on the research questions, the outcomes of Chapter 5 reflect those of Chapter 4: the mainstream coroutine solutions can be applied effectively on these mid-size platforms (RQ1), the costs – and benefits – of using coroutines are found to be deterministic (RQ2) and the benefits are clearly demonstrated by the test results (RQ3).

6.2.5 Research question conclusions

RQ1: Can mainstream coroutine solutions apply to resource-constrained platforms?

In summary, this thesis concludes that, while there is a significant demand for mainstream C++ coroutines on smaller platforms such as microcontrollers and edge devices, problems contained within the current standard continue to be likely to prevent their widespread adoption for embedded and IoT programming.

However, the mainstream specifications and their implementations were found to effective when applied to mid-sized platforms such as those used in Chapters 4 and 5.

RQ2: Are the costs deterministic?

The thesis notes that, while the costs in terms of the C++20 coroutine implementations are deterministic with regard to both memory and speed of performance, the current tool-chains are unable to report memory costs in advance, and that minor changes to the tool-chains would be required in order to address this efficiently.

RQ3: Can the benefits be clearly demonstrated?

Finally, we note that despite the non-optimal outcomes with regard to research questions 1 and 2, there are already clearly demonstrable benefits to the use of C++20 standard coroutines on both microcontrollers and edge devices.

On a microcontroller, we observed substantial improvements in both speed and memory usage when coroutines were used in place of standard real-time operating systems' threads; and when coroutines replaced Duff's Device techniques such as Protothreads, we saw an immediate improvement in code clarity, quality and maintainability without a significantly increased cost in terms of time or memory use.

Furthermore, when C++ native coroutines were used to manage the memory access patterns of highly iterative operations on low-powered edge devices, we saw substantial improvements not only in speed of performance – up to 20% – but also in power use, offering net energy use benefits of 13% to 18% on the platforms tested.

6.3 Contributions

The contributions of this work can be summarised as follows:

- We exhaustively analysed the academic literature through a systematic review and demonstrated a demand for mainstream, language-native coroutine implementations that were suited to resource-constrained platforms.
- We demonstrated that the benefits of mainstream C++20 coroutines are already demonstrable on smaller platforms, but are not fully available in environments where either exceptions or dynamic memory allocation are unacceptable.
- For platforms where C++ exceptions are not acceptable, we developed a standard implementation of the C++20 coroutine definition that avoided the use of exceptions.
- For platforms where dynamic memory allocation is to be avoided, we offered a technique within the current standard for using coroutines without risking heap allocations, and proposed enhancements to the tool-chain that would greatly simplify the use of such techniques.

- We tested a C++20 coroutine implementation on a low-end ARM Cortex-M4 microcontroller and demonstrated that coroutines offered substantially improved memory use and speed of execution when compared to standard real-time operating systems' threads.
- Through exhaustive and repeatable experiments on a variety of edge devices, we demonstrated that the use of C++20 coroutines for several algorithms typically performed on edge device tasks could improve execution speed without substantially increasing code complexity, showing speed boosts of up to 12%, 15.5%, 34% and 65% on the various algorithms. We explored a large parameter space to show that these savings could be achieved for a wide variety of task types and sizes.
- Furthermore, we demonstrated experimentally that C++20 coroutine use in a practical application could not only reduce performance duration but also reduce the power demands of an iterative data processing task on an edge device by up to 13% and 18% (on Raspberry Pi 3 and 4 platforms respectively).

6.4 Further work

To usefully continue the contribution of this thesis, the following further work is suggested:

- Develop and release compiler libraries and/or tool-chain extensions that address the problems raised with regard to RQ2, by:
  - simplifying the management of coroutine stack frames' (CSF) memory location for the programmer (and thereby avoid unintentional heap use);
  - reporting the memory use of CSFs in both stack memory and in global memory;
  - avoiding the use of exceptions in standard library code when exception use is disabled by compiler flags.
- Research the uptake of coroutines in C++, by:
  - Examining the code of C++ open source projects and measuring the adoption of coroutines within projects' code base, and the timescale over which any

adoption has occurred.

  - Surveying developers as to their opinion of the use and usefulness of coroutines.

- The coroutined execution patterns used in chapters 4 and 5 could usefully be developed and investigated further, in the following areas:

  - Test the ease-of-use and reliability of this approach with a panel of C++ programmers who work in the embedded and edge computing sectors.
  - Investigate further the behaviours underlying the detected performance improvements, including an analysis of the impact of the technique on (i) cache misses and stalls and (ii) memory access patterns.
  - Identify other iterative, compute-intensive edge computing tasks that would benefit from the technique, both within and outside ML.
  - Create a standard library in modern C++ that implements the frameworks required for the coroutined execution patterns; and create a compiler extension or preprocessor that automatically translates sequential code to coroutines when requested.
  - Tests of the coroutined execution patterns on smaller, more resource-constrained microcontrollers - such as those used in wireless sensor networks - would extend the generality and usefulness of these results. Even though some low-end microcontrollers lack CPU caches, a positive outcome from the memory access pattern changes alone would be valuable from cost and capacity viewpoints.
  - The experiments in chapters 4 and 5 could usefully be repeated using the newer Raspberry Pi model 5, to investigate how the detected benefits are affected.
  - Investigate the use of coroutines with graphics processing units (GPUs). Where the CPU is responsible for delivering blocks of data to a GPU, there is an opportunity to use coroutines to simplify the code and to improve overall throughput. This was explored before the advent of language-native coroutines [103] and more recently has been applied to register-transfer level (RTL)

simulation on large-scale systems [110]. The emergence of low-power edge devices that include GPUs or tensor processing units offers an opportunity to explore the efficiency of using coroutines for GPU data streaming on resource-constrained devices.

- Encourage the use of coroutines in embedded and edge computing by providing coroutine-based branches of popular machine learning inference libraries for small machines (TinyML). This may result in significant performance gains in both speed and power usage.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (visited on 24/04/2024).

[2] Ala Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys and Tutorials* 17.4 (2015), pp. 2347–2376. ISSN: 1553877X. DOI: 10.1109/COMST.2015.2444095. arXiv: arXiv:1011.1669v3.

[3] Ahmed Ali-Eldin, Bin Wang and Prashant Shenoy. "The Hidden cost of the Edge: A Performance Comparison of Edge and Cloud Latencies". In: *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.* SC '21. New York, NY, USA: Association for Computing Machinery, April 2021, pp. 1–12. ISBN: 9781450384421. DOI: 10.1145/3458817.3476142.

[4] Mariano Alvira and Taylor Barton. "Small and Inexpensive Single-Board Computer for Autonomous Sailboat Control". In: *Robotic Sailing 2012*. Ed. by Colin Sauzé and James Finnis. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2013, pp. 105–116. ISBN: 978-3-642-33084-1. DOI: 10.1007/978-3-642-33084-1_10.

[5] Sidharta Andalam et al. "A Predictable Framework for Safety-Critical Embedded Systems". In: *IEEE Transactions on Computers* 63.7 (2014), pp. 1600–1612. DOI: 10.1109/TC.2013.28.

[6] Michael P. Andersen, Gabe Fierro and David E. Culler. "Enabling synergy in IoT: Platform to service and beyond". In: *Journal of Network and Computer Applications* 81 (2017), pp. 96–110. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.10.017.

[7] Michael P. Andersen, Gabe Fierro and David E. Culler. "System Design for a Synergistic, Low Power Mote/BLE Embedded Platform". In: *Proceedings of the*

2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). Vienna, Austria, April 2016, pp. 1–12. ISBN: 978-1-5090-0802-5. DOI: 10.1109/IPSN.2016.7460722.

[8] Hirochika Asai. "Deep Pipelining: Efficient Pipelining of Network Function Chains with Coroutines". In: *2019 IEEE Conf. on Network Softwarization (NetSoft)*. June 2019, pp. 324–332. DOI: 10.1109/NETSOFT.2019.8806673.

[9] AspenCore. *2017 Embedded Markets Study*. 2017. URL: https://www.eetimes.com/wp-content/uploads/2017-embedded-market-study1-1.pdf (visited on 22/07/2024).

[10] AspenCore. *2019 Embedded Markets Study*. 2019. URL: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf (visited on 22/07/2024).

[11] Luigi Atzori, Antonio Iera and Giacomo Morabito. "The Internet of Things: A survey". In: *Computer Networks* 54.15 (2010), pp. 2787–2805. ISSN: 13891286. DOI: 10.1016/j.comnet.2010.05.010.

[12] Grant Ayers et al. "Classifying Memory Access Patterns for Prefetching". In: *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 513–526. ISBN: 9781450371025. DOI: 10.1145/3373376.3378498.

[13] Roberto Bagnara, Abramo Bagnara and Patricia M. Hill. "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software". In: *International Static Analysis Symposium*. Vol. 11002 LNCS. Springer. 2018, pp. 5–23. ISBN: 9783319997247. DOI: 10.1007/978-3-319-99725-4_2. arXiv: 1809.00821.

[14] Thomas Barnett Jr. et al. *Cisco Global Cloud Index: Forecast and Methodology, 2015–2020*. 2016. URL: https://www.cisco.com/c/dam/m/en_us/service-provider/ciscoknowledgenetwork/files/622_11_15-16-Cisco_GCI_CKN_2015-2020_AMER_EMEAR_NOV2016.pdf (visited on 24/04/2024).

[15] Richard Barry. *The FreeRTOS Kernel*. 2018. URL: https://www.freertos.org/RTOS.html (visited on 02/02/2018).

[16] Bruce Belson and Bronson Philippa. "Speeding up Machine Learning Inference on Edge Devices by Improving Memory Access Patterns using Coroutines". In: *2022 IEEE 25th International Conference on Computational Science and Engineering (CSE)*. IEEE, December 2022, pp. 9–16. ISBN: 979-8-3503-9633-1. DOI: 10.1109/CSE57773.2022.00011.

[17] Bruce Belson et al. "A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 18.3 (June 2019), 21:1–21:21. ISSN: 1539-9087. DOI: 10.1145/3319618.

[18] Bruce Belson et al. "C++20 Coroutines on Microcontrollers—What We Learned". In: *IEEE Embedded Syst. Lett.* 13.1 (2021), pp. 9–12. DOI: 10.1109/LES.2020.2973397.

[19] Alexandre Bergel et al. "FlowTalk: Language Support for Long-Latency Operations in Embedded Devices". In: *IEEE Transactions on Software Engineering* 37.4 (2011), pp. 526–543. ISSN: 00985589. DOI: 10.1109/TSE.2010.66.

[20] Gavin Bierman et al. "Pause 'n' Play: Formalizing Asynchronous C#". In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–257. ISBN: 978-3-642-31057-7. DOI: 10.1007/978-3-642-31057-7_12.

[21] Nicholas M. Boers et al. "Developing wireless sensor network applications in a virtual environment". In: *Telecommunication Systems* 45.2-3 (2010), pp. 165–176. ISSN: 10184864. DOI: 10.1007/s11235-009-9246-x.

[22] Amirali Boroumand et al. "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks". In: *Proc. 30th Int. Conf. Parallel Architectures Compilation Techn. (PACT)*. IEEE, 2021, pp. 159–172. ISBN: 9781665442787. DOI: 10.1109/PACT52795.2021.00019. arXiv: 2109.14320.

[23] Pearl Brereton et al. "Lessons from applying the systematic literature review process within the software engineering domain". In: *Journal of Systems and Software* 80.4 (2007), pp. 571–583. ISSN: 01641212. DOI: 10.1016/j.jss.2006.07.009.

[24] Etienne Brodu, Stéphane Frénot and Frédéric Oblé. "Toward Automatic Update from Callbacks to Promises". In: *Proceedings of the 1st Workshop on All-Web Real-Time Systems - AWeS '15*. New York, New York, USA: ACM Press, 2015, pp. 1–8. ISBN: 9781450334778. DOI: 10.1145/2749215.2749216.

[25] Peter A. Buhr and Ashif S. Harji. "Concurrent urban legends". In: *Concurrency Computation Practice and Experience* 17.9 (2005), pp. 1133–1172. DOI: 10.1002/cpe.885.

[26] Alejandro Cartas et al. "A Reality Check on Inference at Mobile Networks Edge". In: *Proc. of the 2nd Int. Workshop on Edge Systems, Analytics and Networking*. 2019, pp. 54–59. ISBN: 9781450362757. DOI: 10.1145/3301418.3313946.

[27] Rebecca Carter, Andrew Cruden and Peter J. Hall. "Optimizing for Efficiency or Battery Life in a Battery/Supercapacitor Electric Vehicle". In: *IEEE Trans. Veh. Technol.* 61.4 (2012), pp. 1526–1533. DOI: 10.1109/TVT.2012.2188551.

[28] Yunda Chai et al. "Implementation and Optimization of Data Prefetching Algorithm Based on LLVM Compilation System". In: *J. Phys.: Conf. Ser.* 1827.1 (2021). DOI: 10.1088/1742-6596/1827/1/012136.

[29] Zhuoqing Chang et al. "A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things". In: *IEEE Internet Things J.* 8.18 (2021), pp. 13849–13875. DOI: 10.1109/JIOT.2021.3088875.

[30] Shimin Chen et al. "Improving hash join performance through prefetching". In: *ACM Trans. Database Syst.* 32.3 (2007), 17–es. ISSN: 03625915. DOI: 10.1145/1272743.1272747.

[31] Yu Hsin Chen et al. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE J. Emerg. Sel. Top. Circuits Syst.* 9.2 (2019), pp. 292–308. ISSN: 21563365. DOI: 10.1109/JETCAS.2019.2910232. arXiv: 1807.07928.

[32] Eric Chung et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38.2 (2018), pp. 8–20. ISSN: 02721732. DOI: 10.1109/MM.2018.022071131.

[33] David L. Clark. "Powering intelligent instruments with Lua scripting". In: *2009 IEEE AUTOTESTCON*. IEEE, September 2009, pp. 101–106. ISBN: 978-1-4244-4980-4. DOI: 10.1109/AUTEST.2009.5314042.

[34] Marcelo Cohen et al. "Using Coroutines for RPC in Sensor Networks". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–8. ISBN: 1424409101. DOI: 10.1109/IPDPS.2007.370458.

[35] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018. DOI: 10.1201/9780429489105. arXiv: 1709.00084.

[36] Melvin E Conway. "Design of a separable transition-diagram compiler". In: *Commun. ACM* 6.7 (July 1963), pp. 396–408. ISSN: 00010782. DOI: 10.1145/366663.366704.

[37] Nitin Dahad. *The Current State of Embedded Development*. 2023. URL: https://www.embedded.com/embedded-survey-2023-more-ip-reuse-as-workloads-surge/ (visited on 14/04/2022).

[38] Ole Johan Dahl and Kristen Nygaard. "SIMULA: An ALGOL-based simulation language". In: *Communications of the ACM* 9.9 (1966), pp. 671–678. ISSN: 15577317. DOI: 10.1145/365813.365819.

[39] Robert David et al. "Tensorflow lite micro: Embedded machine learning for TinyML systems". In: *Proc. Mach. Learn. Syst.* 3 (2021), pp. 800–811.

[40] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting coroutines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.2 (February 2009), pp. 1–31. ISSN: 01640925. DOI: 10.1145/1462166.1462167.

[41] Ana Lúcia De Moura, Noemi Rodriguez and Roberto Ierusalimschy. "Coroutines in Lua". In: *Journal of Universal Computer Science* 10.7 (July 2004), pp. 910–925. ISSN: 0958695X. DOI: 10.3217/jucs-010-07-0910.

[42] Thierry Delisle. "Concurrency in C∀". PhD thesis. University of Waterloo, Waterloo, ON Canada, 2018.

[43] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. IEEE, June 2009, pp. 248–255. ISBN: 978-1-4244-3992-8. DOI: 10.1109/CVPR.2009.5206848.

[44] Peter J Denning. "The Locality Principle". In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782. DOI: 10.1145/1070838.1070856.

[45] John Donnal. "Joule: A Real Time Framework for Decentralized Sensor Networks". In: *IEEE Internet Things J.* 5.5 (October 2018), pp. 3615–3623. ISSN: 23274662. DOI: 10.1109/JIOT.2018.2815432.

[46] Tom Duff. *Duff's Device*. 1988. URL: https://www.lysator.liu.se/c/duffs-device.html (visited on 19/06/2017).

[47] A. Dunkels, B. Gronvall and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors". In: *29th Annual IEEE International Conference on Local Computer Networks*. IEEE (Comput. Soc.), 2004, pp. 455–462. ISBN: 0-7695-2260-2. DOI: 10.1109/LCN.2004.38.

[48] Adam Dunkels. *About protothreads*. 2005. URL: http://dunkels.com/adam/pt/about.html (visited on 31/01/2018).

[49] Adam Dunkels et al. "Protothreads". In: *Proceedings of the 4th international conference on Embedded networked sensor systems - SenSys '06*. Ed. by '. New York, New York, USA: ACM Press, 2006, p. 29. ISBN: 1595933433. DOI: 10.1145/1182807.1182811.

[50] Caglar Durmaz et al. "Modelling Contiki-Based IoT Systems". In: *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*. Ed. by Ricardo Queirós et al. Vol. 56. OpenAccess Series in Informatics (OASIcs) 5. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 5:1–5:13. ISBN: 978-3-95977-056-9. DOI: 10.4230/OASIcs.SLATE.2017.5.

[51] ECMA. *ECMAScript Latest Draft (ECMA-262) Async Function Definitions*. 2017. URL: https://tc39.github.io/ecma262/#sec-async-function-definitions (visited on 23/07/2024).

[52] Jonathan Edwards. "Coherent reaction". In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*. New York, New York, USA: ACM Press, 2009, pp. 925–932. ISBN: 9781605587684. DOI: 10.1145/1639950.1640058.

[53] Mohammed S. Elbamby et al. "Wireless Edge Computing With Latency and Reliability Guarantees". In: *Proc. IEEE* 107.8 (2019), pp. 1717–1737. ISSN: 15582256. DOI: 10.1109/JPROC.2019.2917084. arXiv: 1905.05316.

[54] Roman Elizarov et al. "Kotlin coroutines: Design and implementation". In: *Onward! 2021 - Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2021* (2021), pp. 68–84. DOI: 10.1145/3486607.3486751.

[55] Atis Elsts et al. "Internet of Things for smart homes: Lessons learned from the SPHERE case study". In: *2017 Global Internet of Things Summit (GIoTS)*. 2017, pp. 1–6. ISBN: 9781509058730. DOI: 10.1109/GIOTS.2017.8016226.

[56] Murali Emani et al. "A Comprehensive Evaluation of Novel AI Accelerators for Deep Learning Workloads". In: *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2022, pp. 13–25. ISBN: 9781665451857. DOI: 10.1109/PMBS56514.2022.00007.

[57] Ralf S Engelschall. "Portable Multithreading-The Signal Stack Trick for User-Space Thread Creation". In: *USENIX Annual Technical Conference, General Track*. 2000.

[58] L Evers et al. "SensorScheme: Supply chain management automation using Wireless Sensor Networks". In: *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. September 2007, pp. 448–455. DOI: 10.1109/EFTA.2007.4416802.

[59] Gabriel Falcao and João Dinis Ferreira. "To PiM or Not to PiM". In: *Commun. ACM* 66.6 (2023), pp. 48–55. ISSN: 15577317. DOI: 10.1145/3589995.

[60] Rene Fritzsche and Christian Siemers. "Scheduling of Time Enhanced C (Tec)". In: *2010 World Automation Congress*. 2010, pp. 1–6. ISBN: 9781424496730.

[61] Mingyu Gao et al. "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory". In: *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)* 52.4 (2017), pp. 751–764. DOI: 10.1145/3037697.3037702.

[62] David Gay et al. "The nesC Language: A Holistic Approach to Networked Embedded Systems". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03*. New York, New York, USA: ACM Press, 2003, pp. 1–11. ISBN: 1581136625. DOI: 10.1145/781131.781133.

[63] Damien George. *MicroPython - Python for microcontrollers*. 2014. URL: http://micropython.org/ (visited on 31/07/2017).

[64] Damien P. George and Paul Sokolovsky. *General information about the ESP8266 port — MicroPython 1.9.4 documentation*. 2014. URL: http://docs.micropython.org/en/latest/esp8266/esp8266/general.html (visited on 04/09/2018).

[65] Christina Giannoula et al. "Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures". In: *SIGMETRICS Perform. Eval. Rev.* 50.1 (2022), pp. 33–34. ISSN: 0163-5999. DOI: 10.1145/3547353.3522661.

[66] Michael Gibbs and Bjarne Stroustrup. "Fast dynamic casting". In: *Software - Practice and Experience* 36.2 (2006), pp. 139–156. DOI: 10.1002/spe.686.

[67] R. Glistvain and M. Aboelaze. "Romantiki OS - A single stack multitasking operating system for resource limited embedded devices". In: *Informatics and Systems (INFOS), 2010 The 7th International Conference on*. 2010, pp. 1–8. ISBN: 978-1-4244-5828-8.

[68] Lois Goldthwaite. "Technical report on C++ performance". In: *ISO/IEC PDTR* 18015 (2006).

[69] Nat Goodspeed. "Stackful Coroutines and Stackless Resumable Functions. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper N4232". 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf.

[70] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. ISSN: 0167739X. DOI: 10.1016/j.future.2013.01.010. arXiv: 1207.0203.

[71] Rajesh K Gupta, Claudionor Nunes Coelho Jr. and Giovanni De Micheli. "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components". In: *Proceedings of the 29th ACM/IEEE Design Automation*

Conference. DAC '92. DAC '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 225–230.

[72] Niklas Gustafsson. "Resumable Functions. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper N3328." 2012. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3328.pdf.

[73] Reinhard von Hanxleden. "SyncCharts in C: A Proposal for Light-weight, Deterministic Concurrency". In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. New York, NY, USA: ACM, 2009, pp. 225–234. ISBN: 978-1-60558-627-4. DOI: 10.1145/1629335.1629366.

[74] Charles R Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (September 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[75] Yongjun He, Jiacheng Lu and Tianzheng Wang. "Corobase: Coroutine-oriented main-memory database engine". In: *Proc. VLDB Endow.* 14.3 (2020), pp. 431–444. ISSN: 21508097. DOI: 10.14778/3430915.3430932. arXiv: arXiv:2010.15981v1.

[76] Ralph Hempel. "Porting Lua to a microcontroller". In: *Lua Programming Gems*. Lua.org, 2008. Chap. 26, pp. 313–324. ISBN: 8590379841.

[77] Dominic Herity. *Modern C++ in embedded systems – Part 1: Myth and Reality — Embedded*. 2015. URL: https://www.embedded.com/modern-c-in-embedded-systems-part-1-myth-and-reality/ (visited on 24/07/2024).

[78] Rui Hu et al. "A survey on data provenance in IoT". In: *World Wide Web* 23.2 (2020), pp. 1441–1463. DOI: 10.1007/s11280-019-00746-1.

[79] R Inam et al. "Support for hierarchical scheduling in FreeRTOS". In: *ETFA2011*. September 2011, pp. 1–10. DOI: 10.1109/ETFA.2011.6059016.

[80] ISO/IEC. "C++ Draft International Standard - N4860". 2020. URL: https://isocpp.org/files/papers/N4860.pdf.

[81] ISO/IEC. "N4680 Programming Languages - C++ Extensions for Coroutines". 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf.

[82] ISO/IEC. "N4775: Working Draft, C++ Extensions for Coroutines". 2018. URL: https://isocpp.org/files/papers/n4775.pdf.

[83] Pekka Jääskeläinen et al. "Reducing Context Switch Overhead with Compiler-Assisted Threading". In: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. Vol. 2. 2008, pp. 461–466. ISBN: 9780769534923. DOI: 10.1109/EUC.2008.181.

[84] Erwan Jahier. "RDBG: A Reactive Programs Extensible Debugger". In: *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*. Ed. by Sander Stuijk. SCOPES '16. New York, NY, USA: ACM, 2016, pp. 116–125. ISBN: 978-1-4503-4320-6. DOI: 10.1145/2906363.2906372.

[85] Christopher Jonathan et al. "Exploiting coroutines to attack the "killer nanoseconds"". In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1702–1714. ISSN: 21508097. DOI: 10.14778/3236187.3236216.

[86] Norman P Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246.

[87] Rubem Kalebe, Gustavo Girao and Itamir Filho. "A library for scheduling lightweight threads in Internet of Things microcontrollers". In: *2017 International Conference on Computing Networking and Informatics (ICCNI)*. IEEE, October 2017, pp. 1–7. ISBN: 978-1-5090-4642-3. DOI: 10.1109/ICCNI.2017.8123793.

[88] Kennedy Kambona, Elisa Gonzalez Boix and Wolfgang De Meuter. "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. DYLA '13. New York, NY, USA: ACM, 2013, 3:1–3:9. ISBN: 978-1-4503-2041-2. DOI: 10.1145/2489798.2489802.

[89] Marcin Karpinski and Vinny Cahill. "High-Level Application Development is Realistic for Wireless Sensor Networks". In: *2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. 2007, pp. 610–619. DOI: 10.1109/SAHCN.2007.4292873.

[90] Andreas Kärrby. *Evaluating Swift concurrency on the iOS platform: A performance analysis of the task-based concurrency model in Swift 5.5*. 2022. URL: https://www.diva-portal.org/smash/get/diva2:1711132/FULLTEXT02.

[91] Meysam Khezri, Mehdi Agha Sarram and Fazlollah Adibniya. "Simplifying Concurrent Programming of Networked Embedded Systems". In: *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. 2008, pp. 993–998. ISBN: 978-0-7695-3471-8. DOI: 10.1109/ISPA.2008.138.

[92] Vladimir Kiriansky et al. "Cimple: instruction and memory level parallelism: a DSL for uncovering ILP and MLP". In: *Proc. of the 27th Int. Conf. on Parallel Architectures and Compilation Techniques*. Limassol Cyprus: ACM, November 2018, pp. 1–16. ISBN: 978-1-4503-5986-3. DOI: 10.1145/3243176.3243185.

[93] Barbara Kitchenham. "Procedures for performing systematic reviews". In: *Keele, UK, Keele University*. Vol. 33. TR/SE-0401. 2004, pp. 1–26.

[94] Barbara Kitchenham and Stuart Charters. "Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3". In: *Technical Report EBSE-2007-01*. Keele University & University of Durham, 2007. ISBN: 1595933751.

[95] Barbara A. Kitchenham, David Budgen and O. Pearl Brereton. "Using mapping studies as the basis for further research - A participant-observer case study". In: *Information and Software Technology* 53.6 (2011), pp. 638–651. ISSN: 09505849. DOI: 10.1016/j.infsof.2010.12.011.

[96] Donald E Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms (3rd. ed.)* Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[97] Christopher Kohlhoff. "A Universal Model for Asynchronous Operations. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper N3747". 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf.

[98] Oliver Kowalke and Nat Goodspeed. "A low-level API for stackful context switching. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper N4397". 2015. URL: https://www.open-std.org/Jtc1/sc22/WG21/docs/papers/2015/p0099r0.pdf.

[99] Oliver Kowalke and Nat Goodspeed. "A proposal to add coroutines to the C++ standard library. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper N3708." 2013. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3708.pdf.

[100] Patrick Kugler, Philipp Nordhus and Bjoern Eskofier. "Shimmer, Cooja and Contiki: A new toolset for the simulation of on-node signal processing algorithms". In: *2013 IEEE International Conference on Body Sensor Networks*. 2013, pp. 1–6. ISBN: 9781479903306. DOI: 10.1109/BSN.2013.6575497.

[101] Nagendra J. Kumar et al. "Efficient Software Implementation of Embedded Communication Protocol Controllers Using Asynchronous Software Thread Integration with Time- and Space-efficient Procedure Calls". In: *ACM Transactions on Embedded Computing Systems* 6.1 (February 2007). ISSN: 1539-9087. DOI: 10.1145/1210268.1210270.

[102] Young Cheon Kwon et al. "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications". In: *Proc. IEEE Int. Solid-State Circuits Conf.* Vol. 64. 2021, pp. 350–352. ISBN: 9781728195490. DOI: 10.1109/ISSCC42613.2021.9365862.

[103] Pavel A Lebedev. "Integrating GPGPU computations with CPU coroutines in C++". In: *Journal of Physics: Conference Series* 681.1 (2016), pp. 1–8. DOI: 10.1088/1742-6596/681/1/012048.

[104] Daewoong Lee et al. "A 16-Gb T-Coil-Based GDDR6 DRAM with Merged-MUX TX, Optimized WCK Operation, and Alternative-Data-Bus Achieving 27-Gb/s/Pin in NRZ". In: *IEEE J. Solid-State Circuits* 58.1 (2023), pp. 279–290. ISSN: 1558173X. DOI: 10.1109/JSSC.2022.3222203.

[105] David Lee and Mihalis Yannakakis. "Principles and methods of testing finite state machines-a survey". In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. ISSN: 00189219. DOI: 10.1109/5.533956.

[106] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition*. MIT Press, 2017. ISBN: 978-0-262-53381-2.

[107] Philip Levis and David Culler. "Maté : A Tiny Virtual Machine for Sensor Networks". In: *ACM SIGPLAN Notices* 37.10 (2002), pp. 85–95. ISSN: 01635964. DOI: 10.1145/605397.605407.

[108] Philip Levis et al. "TinyOS: An Operating System for Sensor Networks". In: *Ambient Intelligence*. Ed. by Werner Weber, Jan M Rabaey and Emile Aarts. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. ISBN: 978-3-540-27139-0. DOI: 10.1007/3-540-27139-2_7.

[109] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. ISSN: 1045-9219. DOI: 10.1109/TPDS.2020.3030548. arXiv: 2002.03794.

[110] Dian-Lun Lin et al. "TaroRTL: Accelerating RTL Simulation Using Coroutine-Based Heterogeneous Task Graph Scheduling". In: *Euro-Par 2024: Parallel Processing*. Ed. by Jesus Carretero et al. Cham: Springer Nature Switzerland, 2024, pp. 151–166. ISBN: 978-3-031-69583-4.

[111] Barbara H Liskov and Liuba Shrira. "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems". In: *ACM SIGPLAN Notices* 23.7 (1988), pp. 260–267. ISSN: 03621340. DOI: 10.1145/960116.54016.

[112] Jialei Liu et al. "Reliability-Enhanced Task Offloading in Mobile Edge Computing Environments". In: *IEEE Internet Things J.* 9.13 (2022), pp. 10382–10396. ISSN: 23274662. DOI: 10.1109/JIOT.2021.3115807.

[113] Weichen Liu et al. "Coroutine-Based Synthesis of Efficient Embedded Software From SystemC Models". In: *IEEE Embedded Systems Letters* 3.1 (2011), pp. 46–49. ISSN: 19430663. DOI: 10.1109/LES.2011.2112634.

[114] LLVM Project. *Download LLVM releases*. 2018. URL: http://releases.llvm.org/ (visited on 14/08/2018).

[115] Daniel Lohmann et al. "The Aspect-Aware Design and Implementation of the CiAO Operating-System Family". In: *Transactions on Aspect-Oriented Software Development IX*. Ed. by Gary T Leavens et al. Berlin, Heidelberg: Springer, 2012, pp. 168–215. ISBN: 978-3-642-35551-6. DOI: 10.1007/978-3-642-35551-6_5.

[116] Magnus Madsen, Ondřej Lhoták and Frank Tip. "A model for reasoning about JavaScript promises". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA, Article 86 (2017), p. 24. ISSN: 24751421. DOI: 10.1145/3133910.

[117] James Manyika et al. *The Internet of Things: Mapping the value beyond the hype*. Tech. rep. June. 2015, pp. 1–24.

[118] Petra Maresova et al. “Consequences of industry 4.0 in business and economics”. In: *Economies* 6.3 Article 46 (2018), pp. 1–14. ISSN: 22277099. DOI: 10.3390/economies6030046.

[119] Christopher D Marlin. “Coroutines: A Programming Methodology, a Language Design and an Implementation”. PhD thesis. University of Adelaide, 1979.

[120] Nicholas D. Matsakis and Felix S. Klock. “The Rust language”. In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.

[121] Erik Meijer. “Reactive extensions (Rx): Curing Your Asynchronous Programming Blues”. In: *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. New York, New York, USA: ACM Press, 2010, p. 1. ISBN: 9781450305167. DOI: 10.1145/1900160.1900173.

[122] Rob van der Meulen. *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. 2017. URL: http://www.gartner.com/newsroom/id/3598917 (visited on 15/08/2017).

[123] Microsoft Corporation. *The latest supported Visual C++ downloads*. 2018. URL: https://support.microsoft.com/en-au/help/2977003/the-latest-supported-visual-c-downloads (visited on 14/08/2018).

[124] Eric Mittelette. *Coroutines in Visual Studio 2015 – Update 1 - Visual C++ Team Blog*. 2015. URL: https://blogs.msdn.microsoft.com/vcblog/2015/11/30/coroutines-in-visual-studio-2015-update-1/ (visited on 04/02/2018).

[125] Aaron Moss, Robert Schluntz and Peter A. Buhr. “C∀: Adding modern programming language features to C”. In: *Software - Practice and Experience* 48.12 (December 2018), pp. 2111–2146. DOI: 10.1002/spe.2624.

[126] Christian Motika and Reinhard von Hanxleden. “Light-weight Synchronous Java (SJL): An approach for programming deterministic reactive systems with Java”. In: *Computing* 97.3 (2015), pp. 281–307. DOI: 10.1007/s00607-014-0416-7.

[127] Todd Mowry and Anoop Gupta. “Tolerating latency through software-controlled prefetching in shared-memory multiprocessors”. In: *J. Parallel Distrib. Comput.* 12.2 (1991), pp. 87–106.

[128] M G Sarwar Murshed et al. "Machine Learning at the Network Edge: A Survey". In: *ACM Comput. Surv.* 54.8 (October 2022), 170:1–170:37. ISSN: 0360-0300. DOI: 10.1145/3469029.

[129] Robert H.B. Netzer and Barton P. Miller. "What Are Race Conditions?: Some Issues and Formalizations". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pp. 74–88. ISSN: 15577384. DOI: 10.1145/130616.130623.

[130] Peter Niebert and Mathieu Caralp. "Cellular Programming". In: *Theory and Practice of Natural Computing, TPNC 2014. Lecture Notes in Computer Science, vol 8890.* Ed. by Adrian-Horia Dediu, Manuel Lozano and Carlos Martín-Vide. Cham: Springer, 2014, pp. 11–22. ISBN: 978-3-319-13749-0. DOI: 10.1007/978-3-319-13749-0_2.

[131] Gor Nishanov. "Incremental Approach: Coroutine TS + Core Coroutines. ISO/IEC JTC1/SC22/WG21 P1362 R0". 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1362r0.pdf.

[132] Gor Nishanov. "Merge Coroutines TS into C++20 working draft. ISO/IEC JTC1/SC22/WG21 P0912 R5". 2019. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0912r5.html.

[133] Gor Nishanov. "Using Coroutine TS with zero dynamic allocations. ISO/IEC JTC1/SC22/WG21 P1365 R0". 2018. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1365r0.pdf.

[134] Uzair A. Noman et al. "From threads to events: Adapting a lightweight middleware for Contiki OS". In: *2017 14th IEEE Annual Consumer Communications and Networking Conference (CCNC)*. 2017, pp. 486–491. ISBN: 9781509061969. DOI: 10.1109/CCNC.2017.7983156.

[135] NXP Semiconductors. *MQX Lite Real-Time Operating System (RTOS)*. 2018. URL: https://www.nxp.com/products/no-longer-manufactured/nxp-mqx-lite-real-time-operating-system-rtos:MQXLITE (visited on 22/04/2019).

[136] Semih Okur et al. "A study and toolkit for asynchronous programming in C#". In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. 1. New York, May 2014, pp. 1117–1127. ISBN: 9781450327565. DOI: 10.1145/2568225.2568309.

[137] Frank Oldewurtel et al. "The RUNES Architecture for Reconfigurable Embedded and Sensor Networks". In: *2009 Third International Conference on Sensor Technologies and Applications*. 2009, pp. 109–116. ISBN: 9780769536699. DOI: 10.1109/SENSORCOMM.2009.26.

[138] Jonathan Paisley and Joseph Sventek. "Real-time Detection of Grid Bulk Transfer Traffic". In: *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*. 2006, pp. 66–72. DOI: 10.1109/NOMS.2006.1687539.

[139] Jianli Pan and James McElhannon. "Future Edge Cloud and Edge Computing for Internet of Things Applications". In: *IEEE Internet Things J.* 5.1 (2018), pp. 439–449. ISSN: 23274662. DOI: 10.1109/JIOT.2017.2767608.

[140] Sihyeong Park et al. "Lua-Based Virtual Machine Platform for Spacecraft On-Board Control Software". In: *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. 2015, pp. 44–51. ISBN: 9781467382991. DOI: 10.1109/EUC.2015.21.

[141] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H Wallach et al. Vol. 32. NeurIPS. Curran Associates, Inc., 2019. arXiv: 1912.01703.

[142] Kai Petersen, Sairam Vakkalanka and Ludwik Kuzniarz. "Guidelines for conducting systematic mapping studies in software engineering: An update". In: *Information and Software Technology* 64 (2015), pp. 1–18. ISSN: 09505849. DOI: 10.1016/j.infsof.2015.03.007.

[143] Kai Petersen et al. "Systematic Mapping Studies in Software Engineering". In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. June. BCS Learning & Development Ltd., 2008, pp. 1–10. DOI: 10.14236/ewic/EASE2008.8.

[144] Aleksandar Prokopec and Fengyun Liu. "Theory and practice of coroutines with snapshots". In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 109. 3. 2018, 3:1–3:32. DOI: 10.4230/LIPIcs.ECOOP.2018.3.

[145] Georgios Psaropoulos et al. "Interleaving with coroutines: a practical approach for robust index joins". In: *Proc. VLDB Endow.* 11.2 (October 2017), pp. 230–242. ISSN: 2150-8097. DOI: 10.14778/3149193.3149202.

[146] Georgios Psaropoulos et al. “Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins”. In: *VLDB J.* 28.4 (2019), pp. 451–471. ISSN: 0949877X. DOI: 10.1007/s00778-018-0533-6.

[147] Al Rainnie and Mark Dean. “Industry 4.0 and the future of quality work in the global digital economy”. In: *Labour & Industry* 30.1 (2020), pp. 16–33. ISSN: 23255676. DOI: 10.1080/10301763.2019.1697598.

[148] Srivaths Ravi et al. “Security in Embedded Systems: Design Challenges”. In: *ACM Transactions on Embedded Computing Systems* 3.3 (2004), pp. 461–491. ISSN: 15583465. DOI: 10.1145/1015047.1015049.

[149] Marcel Rebouças et al. “An Empirical Study on the Usage of the Swift Programming Language”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016* 1 (2016), pp. 634–638. DOI: 10.1109/SANER.2016.66.

[150] Jinke Ren et al. “Collaborative Cloud and Edge Computing for Latency Minimization”. In: *IEEE Trans. Veh. Technol.* 68.5 (2019), pp. 5031–5044. ISSN: 19399359. DOI: 10.1109/TVT.2019.2904244.

[151] Till Riedel et al. “Using web service gateways and code generation for sustainable IoT system development”. In: *2010 Internet of Things (IOT)*. November 2010, pp. 1–8. DOI: 10.1109/IOT.2010.5678449.

[152] Torvald Riegel. “On unifying the coroutines and resumable functions proposals. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper P0073R0”. 2015. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0073r0.pdf.

[153] Dennis M. Ritchie. “The development of the C language”. In: *ACM SIGPLAN Notices* 28.3 (1993), pp. 201–208. ISSN: 15581160. DOI: 10.1145/155360.155580.

[154] Geoff Romer, James Dennett and Chandler Carruth. “Core Coroutines: Making coroutines simpler, faster, and more general. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper P1063R0.” 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1063r0.pdf.

[155] Geoffrey Romer et al. "Coroutines: Use-cases and Trade-offs. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper P1493R0." 2019. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1493r0.pdf.

[156] Silvana Rossetto and Noemi Rodriguez. "A cooperative multitasking model for networked sensors". In: *Proc. 26th Int. Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW'06)* (July 2006), p. 91. ISSN: 15450678. DOI: 10.1109/ICDCSW.2006.5.

[157] Guido van Rossum and Phillip J. Eby. "PEP 342 – Coroutines via Enhanced Generators". 2005. URL: https://www.python.org/dev/peps/pep-0342/.

[158] Peter J. Rousseeuw and Christophe Croux. "Alternatives to the Median Absolute Deviation". In: *J. Am. Stat. Assoc.* 88.424 (1993), pp. 1273–1283. ISSN: 1537274X. DOI: 10.1080/01621459.1993.10476408.

[159] Miguel A Rubio, Carolina Mañoso Hierro and Ángel Pérez de Madrid y Pablo. "Using Arduino To Enhance Computer Programming Courses in Science and Engineering". In: *Proceedings of the EDULEARN13* 72.July (2013), pp. 5127–5133. ISSN: 2340-1117.

[160] Farzad Samie, Lars Bauer and Jörg Henkel. "From Cloud Down to Things: An Overview of Machine Learning in Internet of Things". In: *IEEE Internet Things J.* 6.3 (June 2019), pp. 4921–4934. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2893866.

[161] Mahadev Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Comput.* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

[162] Paul H Schimpf. "Modified protothreads for embedded systems". In: *J. Comput. Sci. Coll.* 28.1 (October 2012), pp. 177–184. ISSN: 1937-4771.

[163] Nikolaos Schizas et al. "TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review". In: *Future Internet* 14.12 Article 363 (2022). ISSN: 19995903. DOI: 10.3390/fi14120363.

[164] Temitayo Shenkoya. "Social change: A comparative analysis of the impact of the IoT in Japan, Germany and Australia". In: *Internet of Things* 11 (2020). ISSN: 25426605. DOI: 10.1016/j.iot.2020.100250.

[165] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet Things J.* 3.5 (2016), pp. 637–646. ISSN: 23274662. DOI: 10.1109/JIOT.2016.2579198.

[166] Sabrina Sicari et al. "Security, privacy and trust in Internet of Things: The road ahead". In: *Computer Networks* 76 (January 2015), pp. 146–164. ISSN: 13891286. DOI: 10.1016/j.comnet.2014.11.008.

[167] Stelios Sidiroglou, Giannis Giovanidis and Angelos D Keromytis. "A dynamic mechanism for recovering from buffer overflow attacks". In: *Information Security: 8th International Conference, ISC 2005, Singapore, September 20-23, 2005. Proceedings 8*. Springer. 2005, pp. 1–15.

[168] Ian Skerrett. *IoT Developer Trends 2017 Edition*. 2017. URL: https://ianskerrett.wordpress.com/2017/04/19/iot-developer-trends-2017-edition/ (visited on 12/05/2017).

[169] Steven S. Skiena. "Numerical Problems". In: *The Algorithm Design Manual*. London: Springer London, 2008, pp. 393–433. ISBN: 9781848000698. DOI: 10.1007/978-1-84800-070-4_13.

[170] Alan Jay Smith. "Cache Memories". In: *ACM Comput. Surv.* 14.3 (1982), pp. 473–530. ISSN: 15577341. DOI: 10.1145/356887.356892.

[171] Richard Smith and Gor Nishanov. "Halo: coroutine Heap Allocation eLision Optimization: the joint response (P0981R0)". 2018. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html.

[172] Richard Smith et al. "Coroutines: Language and Implementation Impact. ISO/IEC JTC1/SC22/WG21: C++ Standards Committee paper P1492R0." 2019. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1492r0.pdf.

[173] Vincent St-Amour and Marc Feeley. "PICOBIT: A Compact Scheme System for Microcontrollers". In: *Implementation and Application of Functional Languages. IFL 2009. Lecture Notes in Computer Science, vol 6041.* Ed. by Marco T Morazán and Sven-Bodo Scholz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–17. ISBN: 978-3-642-16478-1. DOI: 10.1007/978-3-642-16478-1_1.

[174] Darko Stefanović et al. “Static code analysis tools: A systematic literature review”. In: *Ann. DAAAM Proc. Int. DAAAM Symp* 31.1 (2020), pp. 565–573. ISSN: 17269679. DOI: 10.2507/31st.daaam.proceedings.078.

[175] Bjarne Stroustrup. “A Set of C++ Classes for Co-routine Style Programming”. MAuuray Hill, New Jersey, 1985. URL: http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release_e/doc/ClassesForCoroutines.pdf.

[176] Bjarne Stroustrup. “An Overview of C++”. In: *SIGPLAN Not.* OOPWORK ’86 October. New York, NY, USA: ACM, 1986, pp. 7–18. ISBN: 0-89791-205-5. DOI: 10.1145/323779.323736.

[177] Bjarne Stroustrup. “An overview of the C++ programming language”. In: *Handbook of Object Technology*. Ed. by Saba Zamir. CRC Press Boca Raton, FL, 1999. ISBN: 0-8493-3135-8.

[178] Bjarne Stroustrup. “Thriving in a crowded and changing world: C++ 2006–2020”. In: *Proceedings of the ACM on Programming Languages* 4.HOPL Article 70 (2020), pp. 1–168. DOI: 10.1145/3386320.

[179] Moritz Strube et al. “Dynamic operator replacement in sensor networks”. In: *The 7th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2010)*. November 2010, pp. 748–750. DOI: 10.1109/MASS.2010.5663821.

[180] Ekawahyu Susilo et al. “A miniaturized wireless control platform for robotic capsular endoscopy using advanced pseudokernel approach”. In: *Sensors and Actuators, A: Physical* 156.1 (2009), pp. 49–58. ISSN: 09244247. DOI: 10.1016/j.sna.2009.03.036.

[181] Ahmet Ali Süzen, Burhan Duman and Betul Şen. “Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn”. In: *Proc. Int. Congr. Hum.-Comput. Interact., Optim. Robot. Appl.* IEEE. 2020, pp. 1–5. ISBN: 9781728193526. DOI: 10.1109/HORA49412.2020.9152915.

[182] Don Syme, Tomas Petricek and Dmitry Lomov. “The F# Asynchronous Programming Model”. In: *Practical Aspects of Declarative Languages. PADL 2011. Lecture Notes in Computer Science* 6539 LNCS (2011), pp. 175–189. DOI: 10.1007/978-3-642-18378-2_15.

[183] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proc. IEEE* 105.12 (2017), pp. 2295–2329. ISSN: 15582256. DOI: 10.1109/JPROC.2017.2761740. arXiv: 1703.09039.

[184] Simon Tatham. *Coroutines in C*. 2000. URL: https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html (visited on 19/06/2017).

[185] The Python Software Foundation. *9.10 Generator Expressions — Python 3.6.15 documentation*. 2019. URL: https://docs.python.org/3.6/tutorial/classes.html#generators (visited on 24/07/2024).

[186] Christian Tismer. *About Stackless*. 2018. URL: https://github.com/stackless-dev/stackless/wiki (visited on 04/09/2018).

[187] Christian Tismer. "Continuations and stackless Python". In: *Proceedings of the 8th International Python Conference*. Vol. 1. 2000.

[188] John Tse and Alan Jay Smith. "CPU cache prefetching: Timing evaluation of hardware implementations". In: *IEEE Trans. Comput.* 47.5 (1998), pp. 509–526. ISSN: 00189340. DOI: 10.1109/12.677225.

[189] UBM Electronics Group. *2015 Embedded Markets Study*. Tech. rep. April. 2015.

[190] UBM TechInsights. *2009 Embedded Market Study*. 2010. URL: https://www.embedded.com/wp-content/uploads/2019/12/2009_embeddedmarketstudy_ubm.pdf (visited on 24/07/2024).

[191] Gaurav Verma et al. "Performance Evaluation of Deep Learning Compilers for Edge Inference". In: *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*. IEEE, 2021, pp. 858–865. ISBN: 9781665435772. DOI: 10.1109/IPDPSW52791.2021.00128.

[192] Melvin M Vopson. "The information catastrophe". In: *AIP Advances* 10.8 (2020). DOI: 10.1063/5.0019941.

[193] Rolf H Weber. "Internet of things: Privacy issues revisited". In: *Computer Law & Security Review* 31.5 (2015), pp. 618–627. DOI: 10.1016/j.clsr.2015.07.002.

[194] Elecia White. *Making embedded systems, 2nd Ed.* O'Reilly Media, Inc., 2024. ISBN: 9781098151546.

[195] Geoffrey X. Yu et al. "Habitat: A Runtime-Based computational performance predictor for deep neural network training". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 503–521. arXiv: 2102.00527.

[196] Min Yu, SiJi Xiahou and XinYu Li. "A Survey of Studying on Task Scheduling Mechanism for TinyOS". In: *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing* (2008), pp. 1–4. DOI: 10.1109/WiCom.2008.960.

[197] Zeming Yu, Linhai Song and Yiying Zhang. "Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software". In: *arXiv preprint arXiv:1902.01906* (2019). arXiv: 1902.01906.

[198] Jiale Zhang et al. "Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues". In: *IEEE Access* 6 (2018), pp. 18209–18237. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2820162.

[199] Hongbin Zheng et al. "Optimizing Memory-Access Patterns for Deep Learning Accelerators". In: *arXiv preprint arXiv:2002.12798* (2020). arXiv: 2002.12798.

[200] Ian Zhou et al. "Internet of Things 2.0: Concepts, Applications, and Future Directions". In: *IEEE Access* 9 (2021), pp. 70961–71012. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3078549.