Input-centric profiling and prediction for computational offloading of mobile applications

PhD Thesis submitted by

Adam James Rehn

BIT (Hons)

For the degree of Doctor of Philosophy

College of Business, Law and Governance

James Cook University

Cairns, Queensland 4870

Australia

December, 2016

Acknowledgements

This thesis represents the culmination of a doctoral journey that simply would not have been possible without the support of some truly wonderful people. First and foremost, I would like to thank my supervisors for their guidance and their patience. I thank my primary supervisor, Dr Jason Holdsworth, for his years of enthusiasm and encouragement. Throughout the course of my undergraduate and postgraduate education, Jason has acted as a sounding board for countless ideas, and served as a source of both wisdom and inspiration. I thank my secondary supervisor, Professor John Hamilton, for his insights into academia and the research process, his mentorship, and for his unwavering belief in me. I thank my associate supervisor, Dr Singwhat Tee, for his guidance on statistical methods and experimental design. I thank my other associate supervisor, Professor Ickjai Lee, for his insights into the field of data mining. I would particularly like to thank Jason, John, and Singwhat for the tremendous support they provided in helping me navigate through the final year of my doctorate.

I would like to thank my family for their love and support throughout my entire educational journey, and in particular for their patience and understanding throughout the sometimes stressful doctoral experience.

Special thanks go to Dr Leonie Cassidy for her valuable insights into the Design Science Research Methodology, to Aidan Possemiers, for providing his expertise on parallel programming using CUDA, and to Dr Bronson Phillipa, for providing insights into the configuration and performance monitoring of embedded systems devices. I would also like to thank my fellow PhD students and graduates at James Cook University for sharing their experiences as we each undertook our own doctoral journeys.

Thanks to Andrew Joy, Matthew Joy, and Sarah Krivan, for their logistical assistance in running data collection on 20 iPad Air devices.

I would also like to acknowledge the educational program of Apple, Inc., for providing access to the 20 iPad Air devices used for data collection in Chapter 3 and Chapter 7.

Statement of the Contribution of Others

Research Funding

Australian Postgraduate Award (APA) Research Scholarship stipend 3.5 years

College of Business, Law and Governance Graduate Research Scheme award

Thesis Committee

Dr Jason Holdsworth, Information Technology, James Cook University

Professor John Hamilton, Management and Governance, James Cook University

Professor Ickjai Lee, Information Technology, James Cook University

Dr SingWhat Tee, Accounting and Finance, James Cook University

Statistical Support

Professor John Hamilton

Dr Singwhat Tee

Editorial Support

Dr Jason Holdsworth

Professor John Hamilton

Professor Ickjai Lee

Abstract

Relentless growth in mobile computing technologies continues to bring the vision of ubiquitous computing closer than ever before. As mobile hardware becomes more powerful and connected, user expectations of smartphones and tablets continue to push the boundaries of device capabilities. However, the inherent limitations of portability continue to impose constraints that are not present on traditional server platforms. The confluence of high-speed network connections and cloud compute services allows mobile applications to leverage cloud servers, in order to augment device capabilities and overcome the limitations of mobility.

Computational offloading frameworks are a widely-researched tool for optimising the performance and resource use of mobile applications by exploiting the benefits of cloud computing. Offloading frameworks transparently facilitate the delegation of processing to remote servers, allowing mobile applications to utilise the power of cloud computing resources without imposing additional burden on developers. The improved performance and reduced energy use offered by computational offloading helps to mitigate the inherent limitations of mobile devices.

Modern computational offloading frameworks facilitate dynamic offloading decisions that adapt to changing application and device conditions. Adaptive offloading is utilised to ensure that processing is always performed in the most efficient manner. To achieve this, offloading frameworks monitor the behaviour of the application and the state of the mobile device, and perform a cost-benefit analysis designed to produce optimal offloading decisions. The quality of these offloading decisions is dependant on the accuracy of the information used to produce them.

Existing computational offloading frameworks fail to take into account the full influence of an application's input data on its behaviour. Changes in arbitrary input characteristics can result in dramatic changes in application behaviour. However, the overwhelming majority of existing offloading frameworks consider only the size of input data when making offloading decisions. Many frameworks model changes in application behaviour only indirectly through monitoring performance changes over time. This approach assumes a temporal pattern in input characteristics that is not guaranteed to exist, and cannot cleanly differentiate between changes in application behaviour and performance differences caused by changes in device state.

Hardware power saving features implemented in modern mobile devices allow a given device to shift between a number of different system states. Each of these states results in different performance characteristics for mobile applications running on the device. The interactions between hardware components, the mobile operating system, and user-facing applications, result in performance variations that introduce noise into

measurements of application performance. This interference is known as OS noise. Both changes in device state, and the presence of OS noise in performance measurements, must be taken into account by computational offloading frameworks in order to provide accurate information for making offloading decisions.

The overall aim of this thesis is to develop a system for profiling and predicting the performance of mobile applications, that takes into account the influence of arbitrary input characteristics, whilst mitigating the effects of OS noise. This profiling and prediction system should be efficient enough to perform the frequent processing required in order to adapt to changes in device state, and produce accurate predictions of application performance that can be used by a computational offloading framework to make quality offloading decisions.

This thesis focusses on five research objectives in order to address the identified gap in the existing literature. The first three objectives are concerned with expanding the existing understanding of OS noise on traditional desktop and server platforms to encompass mobile device platforms, and examine the interaction between mobile device state changes and OS noise. The fourth objective is concerned with developing a new approach for mitigating the effects of OS noise on performance benchmark data, whilst accounting for the influence of changing device states. The fifth objective applies the outcomes of the preceding objectives to address the overall research aim, and is concerned with developing and input-centric profiling and prediction system for use by a computational offloading framework.

Chapter 1 provides background detail on how computational offloading frameworks function, and presents a thorough literature review of current offloading frameworks. The gap in the existing literature is discussed, and the research question and objectives are described in detail. A brief overview of the Design Science Research Methodology (DSRM) used in this thesis is described, and the choice of methodology is discussed. Overviews of the data chapters of the thesis are provided, along with publication details and a statement of contributions for each chapter.

Chapter 2 describes in detail the DSRM utilised by the research in this thesis. This chapter focusses on extending the existing evaluation methodology of the DSRM to address the unique technical challenges associated with evaluating research outputs that target mobile device platforms. Requirements are drawn from existing literature on both evaluation methodology and mobile data collection, then a model is proposed for automating the evaluation of Design Science artifacts targeting mobile device platforms. The proposed model automates feedback loops of the DSRM to facilitate rapid iteration, providing richer information to researchers.

Chapter 3 addresses research objectives 1, 2, and 3, and addresses part of research objective 4. The first part of this chapter describes a study into the levels of OS noise present on Apple iPad Air devices, which addresses research objective 1. Examination of the collected data confirms that the characteristics of OS noise on mobile devices are consistent with those observed on traditional desktop and server platforms in the existing literature, thus addressing research objective 2. Further data analysis demonstrates that a relationship exists between OS noise levels and changes in device state, thus addressing research objective 3. The second half of the chapter proposes an adaptive noise mitigation technique for mitigating the effects of OS noise on

micro-benchmarking datasets. This addresses the key requirements of research objective 4.

Chapter 4 builds on the work presented in Chapter 3. Research objective 4 requires that the adaptive noise mitigation technique be fully automated. However, the outlier removal phase of the noise mitigation technique described in Chapter 3 is only semi-automated. This chapter proposes a fully automated outlier removal technique. The proposed technique exploits the inherent clustering structure present in outliers introduced by OS noise to perform outlier removal. Hierarchical Agglomerative Clustering (HAC) is used to capture the nested clustering structure of the data, and a heuristic algorithm automatically extracts clusterings that isolate outliers from meaningful data.

Chapter 5 extends the work presented in Chapter 4 by proposing a novel parallel algorithm for performing hierarchical clustering of single-dimensional data. The classical algorithm for HAC features computational costs that make it too expensive for use on resource-constrained mobile devices. The proposed algorithm utilises GPGPU technologies to enable massive parallelism on commodity consumer hardware, including modern mobile devices. By exploiting the unique properties of single-dimensional data, the proposed algorithm maximises the amount of processing that can be performed in parallel. This efficiency makes the proposed algorithm, and thus the automated outlier removal technique proposed in Chapter 4, suitable for use on resource-constrained mobile devices. This addresses the remaining requirements of research objective 4.

Chapter 6 builds the foundation for addressing research objective 5. This chapter reviews existing software timing models from the literature on Worst-Case Execution Time (WCET), and adapts elements of two popular models for use on resource-constrained mobile devices. The proposed timing model simplifies these models to substantially reduce information requirements and computational complexity. Validation of the proposed timing model for individual code execution paths demonstrates that it is an extremely accurate approximation of the information represented by the models that it simplifies. The efficiency of the proposed timing model allows it to be invoked repeatedly to adapt to changes in device state.

Chapter 7 completes the work that was started in Chapter 6 to address research objective 5. This chapter proposes a language- and platform-agnostic tooling pipeline that can be used to implement the proposed input-centric profiling and prediction system for any programming language and mobile device platform. The implementation of a prototype of the proposed model targeting the C++ programming language and the Apple iOS platform is then described. The second half of the chapter validates the predictive power of the input-centric profiling and prediction system for code with multiple execution paths. Validation results demonstrate that the produced predictions are extremely accurate across all of the code modules tested. The accuracy and efficiency of the proposed input-centric performance model make it well-suited for use in a computational offloading framework.

Chapter 8 summarises the outcomes of the preceding chapters, and discusses the contributions of this thesis to the greater body of knowledge. Theoretical, empirical, and real-world contributions are discussed. The limitations of the thesis are then discussed and used to motivate directions for future research.

The outcomes achieved by addressing the five research objectives represent a number of contributions to the greater body of knowledge. The OS noise study undertaken in Chapter 3 to address research objective 1 is

the first detailed exploration OS noise on mobile device platforms of which I am aware. The examination of the collected data that was undertaken to address research objective 2 demonstrates that the characteristics of OS noise on mobile devices are consistent with those characteristics observed on traditional desktop and server platforms in the existing literature. The data analysis performed to address research objective 3 demonstrates that a relationship exists between changes in OS noise levels and changes in device state.

The work done in Chapters 3, 4, and 5 to address research objective 4 resulted in contributions that extend beyond mobile computing, to the field of data mining. The adaptive OS noise mitigation technique proposed in 3 is the first approach for mitigating the effects of OS noise on micro-benchmarking data to maximise accuracy by taking into account the actual noise profile of the device the benchmark is running on. The automated outlier removal technique proposed in Chapter 4 demonstrates that the structure of outliers introduced by OS noise can be exploited to provide an efficient, automated technique for removing such outliers from micro-benchmarking data. The novel GPU-accelerated clustering algorithm proposed in Chapter 5 demonstrates the enormous performance improvements that can be achieved through the maximisation of merge parallelism, by exploiting the unique properties of single-dimensional datasets.

The input-centric profiling and prediction system presented in Chapters 6 and 7 to address research objective 5 represents the key contribution to the field of computational offloading. The application timing model developed in Chapter 6 simplifies popular software timing models from the existing literature to substantially reduce information requirements and computational complexity. This simplification demonstrates the feasibility of adapting timing models for use on resource-constrained mobile devices, even when the unmodified timing models are unsuitable for use in a mobile context.

The language- and platform-agnostic tooling pipeline proposed in Chapter 7 allows the proposed input-centric profiling and prediction system to be implemented for any programming language, and for any target mobile device platform. Evaluation of the input-centric profiling and prediction system in Chapters 6 and 7 demonstrates that the produced predictions are extremely accurate. The proposed system accounts for the influence of arbitrary input characteristics on application behaviour, whilst mitigating the effects of OS noise. The efficiency and accuracy of the proposed system make it well-suited for use in a computational offloading framework. The use of deep application-specific knowledge provides high-quality information for making optimal offloading decisions, thus maximising the performance and energy efficiency of mobile applications.

Table of Contents

Front Matter **i**

Title Page i

Acknowledgements iii

Statement of the Contribution of Others v

Abstract vii

Table of Contents xv

List of Tables xviii

List of Figures xx

List of Abbreviations xxi

1 Introduction **1**

1.1 Background 1

1.1.1 Computational offloading 2

1.1.2 Use of input characteristics in offloading decisions 4

1.1.3 Dynamic performance factors 6

1.1.3.1 Dynamic Voltage and Frequency Scaling 6

1.1.3.2 Operating System Noise 6

1.2 Research Gaps, Questions, and Objectives 7

1.3 Research methodology overview 8

1.4 Thesis Outline 9

2 Research methodology **15**

2.1 Introduction 15

2.2 Background 17

2.2.1 Design science research methodology 17
2.2.1.1 Artifacts 17
2.2.1.2 DSRM Activities 18
2.2.2 Design Science Research Artifact Evaluation 19
2.2.3 Mobile data collection frameworks 21
2.2.3.1 Data collection requirements 22
2.3 Model 24
2.3.1 Objectives and Design Goals 24
2.3.2 Model Architecture 26
2.3.2.1 Mobile Device Middleware 27
2.3.2.2 Server 28
2.3.2.3 Artifact Assessment Deployment System 29
2.4 Model validation 29
2.5 Implications of research 31
2.5.1 Theoretical implications 31
2.5.2 Practical and management implications 32
2.6 Future Research 33
2.7 Conclusion 34

3 Operating System (OS) noise study 35

3.1 Introduction 36
3.2 Background 36
3.2.1 Impact of OS noise within parallel computing 37
3.2.1.1 Sources of OS noise 37
3.2.1.2 Measuring OS noise 39
3.2.2 Micro-benchmark accuracy 41
3.3 Case study: Apple iOS device OS noise profiles 42
3.4 Experimental methodology 43
3.5 Experimental results 44
3.5.1 Outlier removal 44
3.5.2 Data analysis 46

3.6 Adaptive noise mitigation approach 47
3.7 Discussion and future work 49
3.8 Conclusion 50

4 Automated outlier removal **51**

4.1 Introduction 51
4.2 Related work 53
4.3 Data characteristics 54
4.4 Outlier removal algorithm 57
4.5 Simplified heuristic for mobile devices 59
4.5.1 Evaluation 62
4.5.1.1 Outlier removal effectiveness 62
4.5.1.2 Algorithm efficiency 62
4.6 Conclusion 64

5 GPU-accelerated hierarchical clustering **67**

5.1 Introduction 68
5.2 Background 69
5.2.1 Hierarchical Agglomerative Clustering (HAC) 69
5.2.2 General Purpose GPU (GPGPU) languages 70
5.2.3 Parallel implementations of HAC 72
5.3 Implementation 73
5.3.1 Benefits of single-dimensional data 73
5.3.2 Algorithm 74
5.3.3 Handling merge collisions 75
5.3.4 Pre-merging duplicate values 78
5.4 Evaluation 80
5.4.1 Validation of correctness 82
5.4.2 Performance benchmarking 82
5.5 Implications and Future Work 85
5.6 Conclusion 86

6 Input-centric performance model **89**

6.1 Introduction . . . 90

6.2 Background . . . 91

6.2.1 Application profiling . . . 91

6.2.2 Application characteristics . . . 93

6.2.3 Timing models . . . 95

6.3 Model . . . 97

6.4 Validation . . . 102

6.4.1 Synthetic datasets . . . 102

6.4.2 Device datasets . . . 103

6.5 Implications and Future Work . . . 104

6.6 Conclusion . . . 105

7 Input-centric profiling and prediction **107**

7.1 Introduction . . . 107

7.2 Implementation . . . 108

7.2.1 Pipeline section 1: collecting and transforming the execution tree . . . 109

7.2.1.1 Collecting the execution tree . . . 109

7.2.1.2 Transforming the execution tree . . . 111

7.2.2 Pipeline section 2: collecting and transforming the set of instructions . . . 112

7.2.3 Runtime library . . . 112

7.3 Experimental methodology . . . 113

7.4 Results . . . 114

7.5 Implications and Future Work . . . 116

7.6 Conclusion . . . 117

8 Conclusion **119**

8.1 Thesis Outcomes . . . 119

8.1.1 Chapter 2 . . . 119

8.1.2 Chapter 3 . . . 120

8.1.3 Chapter 4 . . . 120

8.1.4 Chapter 5 121

8.1.5 Chapter 6 121

8.1.6 Chapter 7 122

8.2 Theoretical Contributions 123

8.3 Empirical Contributions 124

8.4 Real-world Contributions 125

8.5 Limitations and Recommendations for Future Research 127

References **127**

List of Tables

1.1 Computational offloading frameworks, categorised by the characteristics of input data that they take into account when making offloading decisions. 4

2.1 DSR artifact evaluation dimensions, adapted from Cleven, Gubler and Hüner (2009). Note: vertical lines within dimensions are alternative characteristics or methodological approaches. 20

2.2 Summary of mobile data collection features. 24

3.1 Commonly identified sources of OS noise from the existing literature. Sources can arise from the underlying hardware and operating system, as well as services running on behalf of the operating system. 37

3.2 Common techniques for measuring OS noise from the existing literature. 39

3.3 Mean skewness and kurtosis of the collected data, before and after the removal of outliers. 44

3.4 Comparison of strategies for selecting the value of the scale factor F. 48

4.1 Normalised mean outlier detection error values for the two curves. 61

4.2 Results for the four compactness metrics: TWGD, Depth, Radius, and the difference between the centroid and the medoid. 63

4.3 Time complexity for the full and simplified outlier detection heuristics, excluding dendrogram construction. 63

5.1 A sample of existing approaches for implementing hierarchical clustering in parallel. . . . 72

5.2 Device characteristics for the three GPU models used for benchmarking, as reported by the CUDA *cudaGetDeviceProperties()* function. The value reported for the maximum number of threads is the number of Streaming Multiprocessor (SM) units the device has, multiplied by the maximum number of concurrent threads per SM. 85

6.1 Comparison of existing timing models 96

6.2 Structure of a dataset. Values are examples. 102

6.3 Validation results for real hardware devices 104

7.1 Cross-validation results for the validation experiment, averaged across all 20 devices and all 5 runs of the experiment. 115

List of Figures

1.1 Examples of offloading framework architectures, based on (a) RPC, from MAUI (Cuervo et al., 2010), and (b) VM Migration, from CloneCloud (Chun, Ihm, Maniatis, Naik & Patti, 2011) 3

1.2 Computational offloading cost model, from Kovachev, Cao and Klamma (2011) 3

1.3 Design Science Research Methodology (DSRM) Process Model, adapted from Peffers, Tuunanen, Rothenberger and Chatterjee (2007). 9

2.1 Interactions between the components of my proposed model for automated evaluation of design science research artifacts targeting mobile platforms. 26

2.2 Manual data collection using the Apple Xcode development tool. 30

2.3 DSRM process iteration loops (Peffers et al., 2007) automated by the automated evaluation workflow. 32

3.1 High-level categorisation of OS noise measurement approaches and their outputs. The two main approaches are *user-level micro-benchmarks* and *kernel instrumentation*. The two approaches can be utilised individually or in combination. 41

3.2 Example scatter-plots of the collected micro-benchmarking data before (a) and after (b) outlier removal. Note the difference in scale: 0 - 40,000 for the raw data, 0 - 300 for the data after outlier removal. 46

4.1 Example datasets for each of the mobile device hardware models. The top row depicts each dataset in full, whilst the bottom row depicts each dataset once outliers have been removed. 55

4.2 Example dataset highlighting both inliers and the two types of encountered outliers. 56

4.3 The unique intersections between the ideal ranges of the collected datasets. The y-value and colour saturation of each bar represents the normalised overlap count for that intersection. . 60

4.4 The two cut level ranking curves I evaluate for use with the simplified heuristic. 61

4.5 Log-log plot of the average runtime for the full heuristic and the simplified heuristic, excluding dendrogram construction. 64

5.1 Example of a scenario where merge collisions exist. Only the merges that are not marked as collisions will be merged in parallel by the naive collision detection algorithm, whereas all of the merges marked as “safe to merge” are able to be safely merged in parallel. 76

5.2 Example run of the proposed collision detection algorithm, which maximises merge parallelism. 78

5.3 An example of a scenario where the clustering decisions made by the parallel algorithm can differ from those made by the classical serial algorithm, depending on the properties of the points being clustered. 79

5.4 Example results of the duplicate value pre-processing step. (a) is the sorted list of merges, and (b) is the resulting dendrogram. 81

5.5 Benchmarking results for the CPU and GPU implementations of both complete linkage and single linkage, across the three dataset types. 84

5.6 Benchmarking results for the GPU implementations of both complete linkage and single linkage, across the three dataset types. 84

6.1 Application performance model, expanded from S. Wang, Kodase, Shin and Kiskis (2002) 92

6.2 (a) Code that compares two integers for sorting purposes, and (b) the corresponding Control Flow Graph (CFG) 93

6.3 (a) Code that swaps two integers, and (b) the corresponding symbolic execution tree, from Anand et al. (2013) 94

6.4 Pipeline interference over three nodes and its timing model, from Engblom (2002). 98

6.5 Visualisation of the manner in which the k value for a given execution path is an approximation of the mean of the set W for that execution path. The k value for a path is a reflection of the unique mix of instructions in that path. 101

7.1 High-level overview of the tooling pipeline for collecting and transforming the data required by my input-centric application performance model. 109

7.2 Scatterplots depicting prediction error values on the vertical axis and k-coherence values on the horizontal axis, for each simplification strategy. Each point represents the average value for an individual execution path, averaged across the 20 devices and 5 runs of the experiment. Red lines represent the linear function of best fit for each set of values. 116

List of Abbreviations

Abbreviation	Definition
API	Application Programming Interface
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CUDA	Compute Unified Device Architecture
DSR	Design Science Research
DSRM	Design Science Research Methodology
DVFS	Dynamic Voltage and Frequency Scaling
FTQ	Fixed Time Quantum
FWQ	Fixed Work Quantum
GPGPU	General-Purpose Graphics Processing Unit
GPIO	General-Purpose Input/Output
GPU	Graphics Processing Unit
HAC	Hierarchical Agglomerative Clustering
LOF	Local Outlier Factor
LSQ	Least-Squares
OS	Operating System
RPC	Remote Procedure Call
SA	Simulated Annealing
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
TLB	Translation Lookaside Buffer
TWGD	Total Within-Group Distance
VM	Virtual Machine
WCET	Worst-Case Execution Time

Chapter 1

Introduction

1.1 Background

The domain of mobile and embedded computing continues to experience substantial growth and has established itself as a significant area of research and development (Gartner, Inc., 2016a, 2016b). Mobile computing is now a key component of our modern, information-driven society. Advances in System-on-a-Chip (SoC) hardware manufacturing and wireless networking technologies have resulted in a proliferation of powerful embedded devices with always-on network connectivity. Combined with software techniques for distributed processing, these enabling technologies are driving the realisation of the vision of ubiquitous computing, commonly referred to as the Internet of Things (IoT) (Gubbi, Buyya, Marusic & Palaniswami, 2013).

At the forefront of this mobile revolution are smartphones and tablets. As the most prolific consumer mobile devices, the capabilities of smartphones and tablets stand to have the greatest impact on public perception of mobile computing. As smartphones and tablets become more powerful, users are demanding more from these devices. Graphically and computationally intensive applications such as games continue to be amongst the most popular mobile applications sold (Apple, Inc., 2016; Google, Inc., 2016). However, resources available to mobile devices are always inherently constrained by the requirements of portability. The most notable limitation is battery capacity (Ali, Simoens, Verbelen, Demeester & Dhoedt, 2016). By contrast, servers housed in data-centres and connected to mains power remain more powerful than mobile devices, as they are not constrained by a mobile form factor.

The rise of cloud-based Compute-as-a-Service (CaaS) platforms enables mobile devices to leverage the powerful resources of remote servers (Fernando, Loke & Rahayu, 2013b). The use of remote servers to augment the capabilities of mobile devices has several names, including Mobile Cloud Computing (MCC) (Fernando et al., 2013b; Kovachev et al., 2011; Shiraz, Sookhak, Gani & Shah, 2015) and cyber-foraging (Balan, 2006; Lewis & Lago, 2015; Messinger & Lewis, 2013), but for the sake of simplicity, I will use the term *computational offloading* (Kumar, Liu, Lu & Bhargava, 2013; Yousafzai et al., 2016).

1.1.1 Computational offloading

At its simplest, computational offloading utilises remote code execution to perform processing on a cloud server, and then retrieves the results (Balan, Satyanarayanan, Park & Okoshi, 2003). This allows mobile applications to perform processing that would otherwise be too costly to execute on the mobile device itself. However, applications that utilise unconditional computational offloading cannot function without an active network connection. In addition, there exist circumstances under which remote processing may be detrimental to application performance.

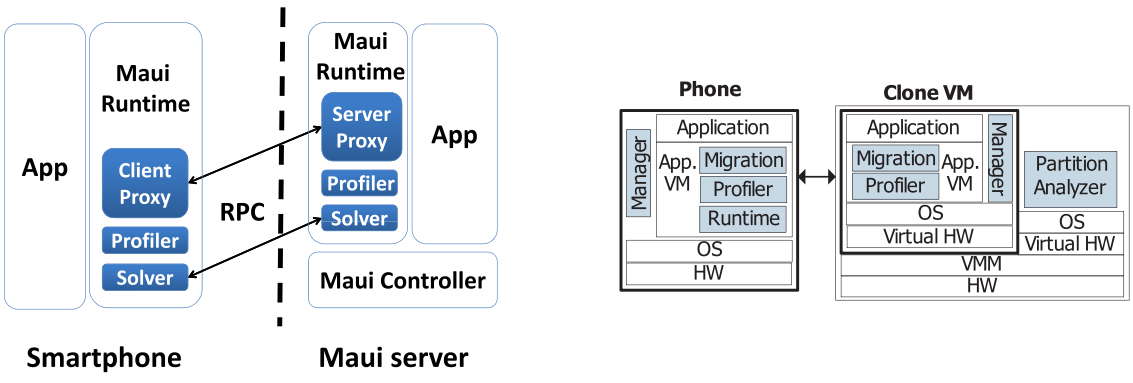
Consider a scenario where a mobile device is connected to a remote server via an extremely slow or congested network connection, and performing execution remotely requires the transmission of a large amount of input and output data. In this case, local execution on the mobile device itself may result in better performance and reduced resource usage when compared to remote execution (Cuervo et al., 2010). Optimal use of computational offloading requires dynamic offloading decisions, informed by the circumstances under which processing takes place. This requirement is addressed by a type of software known as a computational offloading framework.

Computational offloading frameworks are a type of middleware that aim to optimise application performance and resource use through the use of dynamic offloading decisions (Kumar et al., 2013). Offloading frameworks perform two key functions. The first is to make offloading decisions on behalf of the mobile application. The second is to abstract the implementation of migrating application execution between the mobile device and the remote server (Y. Zhang et al., 2012).

The two most common migration mechanisms are Remote Procedure Calls (RPC) (Cidon, London, Katti, Kozyrakis & Rosenblum, 2011; Cuervo et al., 2010; Kemp, Palmer, Kielmann & Bal, 2010; Kosta, Aucinas, Hui, Mortier & Zhang, 2012; Ra et al., 2011; Verbelen, Simoens, Turck & Dhoedt, 2012) and Virtual Machine (VM) Migration (Chun et al., 2011; Flores & Srirama, 2013; Gordon, Jamshidi, Mahlke, Mao & Chen, 2012; Kakadia, Saripalli & Varma, 2013; Satyanarayanan, Bahl, Caceres & Davies, 2009). Regardless of the migration mechanism employed, the high-level architecture of a computational offloading framework remains largely the same. Figure 1.1 depicts two examples of offloading framework architectures, and how they distribute components between a mobile device and a remote server.

Aside from the component responsible for performing application migration, the most common component of a computational offloading framework is the *profiler*. The profiler is responsible for monitoring the condition of the application, mobile device, network connection, and remote server. This information is then used to perform a cost-benefit analysis to determine whether execution of a given part of the application should be offloaded or not (Cuervo et al., 2010). This cost-benefit analysis is commonly formulated as an optimisation problem, which aims to satisfy a set of goals within a given set of constraints. Figure 1.2 depicts the components commonly utilised by the cost model used to make computational offloading decisions.

The cost model may attempt to optimise one or more metrics, such as application performance or power use. The optimisation problem is often solved by feeding the data from the profiler component into a linear solver (Cuervo et al., 2010). The quality of the produced offloading decisions is dependant on the accuracy of the



(a) (b)

Figure 1.1: Examples of offloading framework architectures, based on (a) RPC, from MAUI (Cuervo et al., 2010), and (b) VM Migration, from CloneCloud (Chun et al., 2011)

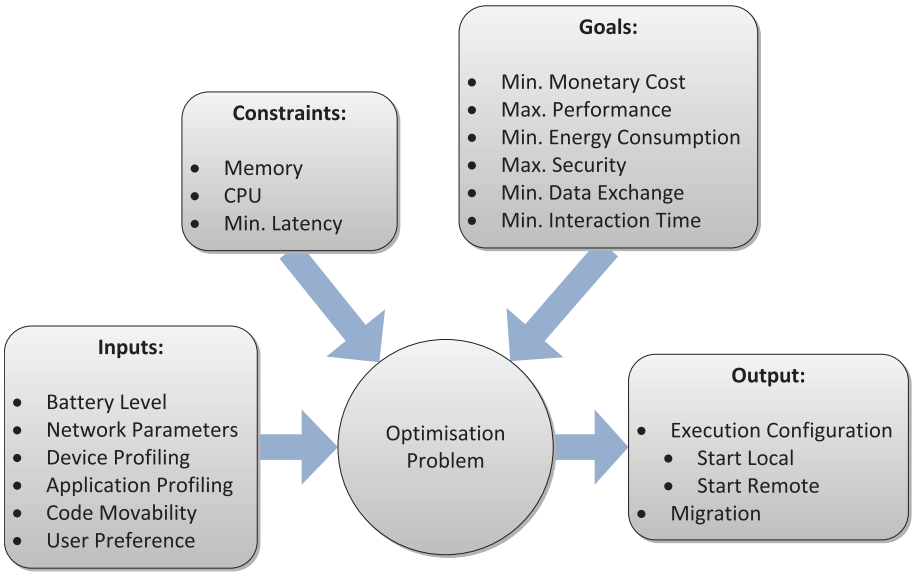


Figure 1.2: Computational offloading cost model, from Kovachev et al. (2011)

information generated by the profiler.

A majority of existing research has focussed on improving the accuracy of the profiled components (Ali et al., 2016; W. Gao, Li, Lu, Wang & Liu, 2014; Gurun, Krintz & Wolski, 2008; Y. Kwon et al., 2013; Pathak, Hu, Zhang, Bahl & Wang, 2011). The two components to receive the most attention are *device profiling* and *application profiling*. Device profiling research has largely focussed on power models for battery consumption of both mobile device hardware components and network data transmission (Ali et al., 2016; Cuervo et al., 2010; Geng, Hu, Yang, Gao & Cao, 2015; B. Gil & Trezentos, 2011). Application profiling has focussed on execution behaviour and memory usage of mobile applications (Chun et al., 2011; W. Gao et al., 2014; C. Wang & Li, 2004; Y. Zhang et al., 2012). Application profiling presents the greater challenge of the two, since a computational offloading framework must be able to adapt to the behaviour patterns of a given mobile application, which are unique to the code in that application.

A number of factors influence application performance. A detailed treatment of these factors can be found in Chapter 6. One factor that can have a significant impact on application behaviour is the application's input

data (W. Gao et al., 2014). The relationship between application input data characteristics and execution behaviour can be entirely arbitrary, and unique to each application (King, 1976). Input characteristics can include the size of the input data (Balan et al., 2003; Cuervo et al., 2010; Flinn, Park & Satyanarayanan, 2002; Ra et al., 2011), the values of individual input parameters (Flores et al., 2015; W. Gao et al., 2014; Shi et al., 2014), or arbitrary features of input data that are acted upon by the application's code (W. Gao et al., 2014; C. Wang & Li, 2004). However, most existing computational frameworks fail to take into account the influence of arbitrary input data characteristics. In Section 1.1.2, I examine the treatment of input data characteristics by existing offloading frameworks, and the extent to which they are factored into making optimal offloading decisions.

1.1.2 Use of input characteristics in offloading decisions

Table 1.1 summarises existing computational offloading frameworks, and categorises them based on the input characteristics that they take into account when making offloading decisions.

Table 1.1
Computational offloading frameworks, categorised by the characteristics of input data that they take into account when making offloading decisions.

Offloading frameworks	Input characteristics taken into account
Goyal and Carter (2004); Gurun et al. (2008); D. Huang, Zhang, Kang and Luo (2010); Huerta-Canepa and Lee (2010); Kemp et al. (2010); March et al. (2011); Ok, Seo and Park (2007); O'Sullivan and Grigoras (2013); Rachuri, Mascolo, Musolesi and Rentfrow (2011); Seshasayee, Nathuji and Schwan (2007); Su and Flinn (2005); Y. Zhang, t. Guan, Huang and Cheng (2009)	None
Noble et al. (1997); Rigole, Berbers and Holvoet (2004)	Programmer-defined
Y.-W. Kwon and Tilevich (2013); O'Hara, Nathuji, Raj, Schwan and Balch (2006); Ra et al. (2011); Rahimi, Venkatasubramanian, Mehrotra and Vasilakos (2012); L. Yang et al. (2013)	Size of data passed between pipeline stages
Chu, Song, Wong, Kurakake and Katagiri (2004); Chun et al. (2011); Cuervo et al. (2010); Duga (2011); Ferrari, Giordano and Puccinelli (2016); Flores and Srirama (2013); B. Gao, He, Liu, Li and Jarvis (2012); Geng et al. (2015); Giurgiu, Riva, Juric, Krivulev and Alonso (2009); Gordon et al. (2012); Gu, Nahrstedt, Messer, Greenberg and Milojicic (2003, 2004); Han, Zhang, Cao, Wen and Zhang (2008); Hung, Shieh and Lee (2012); Hung, Shih, Shieh, Lee and Huang (2012); Hunt and Scott (1999); Imai (2012); Kakadia et al. (2013); Kovachev, Cao and Klamma (2012); Kovachev and Klamma (2012); Li, Wang and Xu (2001); Li and Xu (2002); Ma, Lam and Wang (2011); Ma and Wang (2012); Messer et al. (2002); Messinger and Lewis (2013); Mohapatra and Venkatasubramanian (2003); Nimmagadda, Kumar, Lu and Lee (2010); Ou, Wu, Yang and Zhou (2008); Ou, Yang and Liotta (2006); Ou, Yang and Zhang (2007); Satyanarayanan et al. (2009); Shiraz, Gani, Shamim, Khan and Ahmad (2015); Simanta, Ha, Lewis, Morris and Satyanarayanan (2013); Weinsberg, Dolev, Wyckoff and Anker (2007); Wolski, Gurun, Krintz and Nurmi (2008); K. Yang, Ou and Chen (2008); Yousafzai et al. (2016)	Size of serialised program state
Ahnn and Potkonjak (2013); Ali et al. (2016); Angin and Bhargava (2013); Aucinas, Crowcroft and Hui (2012); Balan et al. (2003); Y.-S. Chang and Hung (2011); G. Chen et al. (2004); X. Chen, Jiao, Li and Fu (2016); Cidon et al. (2011); Dou, Kalogeraki, Gunopulos, Mielikainen and Tuulos (2010); Fernando, Loke and Rahayu (2013a); Flinn et al. (2002); Flores and Srirama (2014); Giurgiu, Riva and Alonso (2012); Hong, Kumar and Lu (2009); Imai and Varela (2011); Kosta et al. (2012); Kremer, Hicks and Rehg (2003); Kristensen (2010); Liu and Lu (2010); Marinelli (2009); Matthews, Chang, Feng, Srinivas and Gerla (2011); Paniagua, Flores and Srirama (2012); Rego et al. (2016); Rim, Kim, Kim and Han (2006); Rong and Pedram (2003); Saab, Saab, Kayssi, Chehab and Elhajj (2015); Shi, Lakafosis, Ammar and Zegura (2012); Srirama, Paniagua and Flores (2012); Verbelen, Hens, Stevens, De Turck and Dhoedt (2010); Verbelen et al. (2012); Verbelen, Stevens, Simoens, Turck and Dhoedt (2011); Xian, Lu and Li (2007); X. Zhang, Jeon, Gibbs and Kunjithapatham (2012); X. Zhang, Kunjithapatham, Jeong and Gibbs (2011); Y. Zhang et al. (2012)	Length of each input parameter
Shi et al. (2014, 2013)	*Key features model* (automatically extracted fine-grained features, such as loop counter values)
Flores et al. (2015)	Input values included in request hashing for memoization
W. Gao et al. (2014)	Features captured by the order-k semi-Markov model
C. Wang and Li (2004)	Arbitrary input characteristics

Some offloading frameworks do not directly take into account input data characteristics. These frameworks utilise approaches that indirectly respond to changes in input characteristics, typically by measuring changes in application performance over time (Goyal & Carter, 2004; Gurun et al., 2008; D. Huang et al., 2010; Kemp

et al., 2010; March et al., 2011; Rachuri et al., 2011). These models do not capture input characteristics themselves, instead capturing only the changes in application behaviour that manifest as a result of changing input data. Predictions generated from these purely history-based models are a function of the time series data they are based upon (Gurun et al., 2008). Accordingly, this approach works only if there exists a temporal pattern in input data characteristics, which is not guaranteed for all mobile applications.

Several frameworks allow the programmer to specify the input characteristics to be taken into account, often by assigning a weighting to each characteristic (Noble et al., 1997; Rigole et al., 2004). This approach is relatively uncommon, as it is cumbersome and places a high level of burden on the application developer.

A number of offloading frameworks model the behaviour of an application as a pipeline of computational stages, with data flowing between them. In such frameworks, the only input characteristic taken into account is the size of the input data and resulting output data flowing between pipeline stages (Y.-W. Kwon & Tilevich, 2013; O'Hara et al., 2006; Ra et al., 2011; Rahimi et al., 2012; L. Yang et al., 2013). This strategy is adapted from traditional distributed computing, where computation is taking place over a series of compute nodes (Ra et al., 2011). This offers increased flexibility, but is not strictly necessary when computation is only performed on one device at a time, which is the most common mode of operation amongst existing offloading frameworks.

Many offloading frameworks utilise a far simpler version of the pipeline-based approach, where communication costs between intermediate stages of the pipeline are ignored because they are known to take place on the same device (Chu et al., 2004; Chun et al., 2011; Cuervo et al., 2010; Gordon et al., 2012; Gu et al., 2003; Satyanarayanan et al., 2009; Weinsberg et al., 2007). This simplified pipeline-based approach closely reflects the VM Migration offloading strategy, whereby the application is migrated back and forth between the mobile device and a server. The only input characteristic taken into account when making offloading decisions is the memory used by input values, as part of the overall size of the application state that must be migrated (Chun et al., 2011; Cuervo et al., 2010; Giurgiu et al., 2009; Han et al., 2008; Satyanarayanan et al., 2009).

Just as one common approach reflects VM Migration, another common approach reflects the use of RPCs. In the RPC-oriented approach, the only characteristic of input data taken into account when making offloading decisions is the length of each input parameter to the offloaded function (Balan et al., 2003; Cidon et al., 2011; Flinn et al., 2002; Kosta et al., 2012; Verbelen et al., 2012). Like the previous approaches that include input data size, the size is used primarily in computing data transfer costs between the mobile device and the server, and less commonly for determining computational cost (Flinn et al., 2002; Imai & Varela, 2011).

Several existing offloading frameworks utilise a model that I refer to as the *key features model* (Shi et al., 2014, 2013). The key features model automatically extracts a set of fine-grained features from an application that can be used to predict the behaviour of the larger application as a whole (L. Huang et al., 2010). Although this model can extract arbitrary features that are indirectly related to input characteristics, prediction is limited to larger parts of the application and the relationship between the input characteristics and the key features themselves cannot always be determined.

The offloading framework proposed by Flores et al. (2015) caches the results of previous computations and

reuses them when possible, a strategy known in computer science as *memoisation*. In order to prevent results from being reused incorrectly for differing input values, the input values themselves are included in the information that is hashed to produce a unique identifier for each cached request. The cached values can also be utilised by multiple mobile devices running the same application (Flores et al., 2015). However, this approach does not incorporate input characteristics into any sort of cost model, it merely utilises caching to provide performance improvements when the same input data is used more than once.

The only existing offloading framework that takes arbitrary input data characteristics into account, that I am aware of, is that proposed by C. Wang and Li (2004). This framework utilises a software analysis technique known as symbolic execution to build a model that maps arbitrary input data characteristics to application behaviours. This represents the greatest extent to which a computational offloading framework can factor input data characteristics into its offloading decisions. However, there are a number of other factors that impact the performance and resource usage of an application at runtime that this framework does not take into account. These factors are discussed in the following section.

1.1.3 Dynamic performance factors

In addition to the application code that is executing and the input data to that code, there are a number of other factors that can impact the performance and resource usage of a mobile application at runtime. The most important of these are Dynamic Voltage and Frequency Scaling (DVFS) and Operating System (OS) noise.

1.1.3.1 Dynamic Voltage and Frequency Scaling

DVFS is a power management technique utilised by modern mobile processors. DVFS dynamically adjusts the voltage and frequency of the processor to optimise power consumption, based on the current processing workload (Mittal, 2014). Similar techniques are employed by other hardware components. The effect of these power management techniques is that a mobile device can transition between multiple steady-states while an application is running.

The existence of multiple device steady-states is of particular importance to computational offloading frameworks because the profiling data recorded during one steady-state cannot be used to accurately predict the performance and resource use of the application during another steady-state. This precludes the use of precomputed training data, and necessitates frequent profiling to ensure prediction accuracy is maintained.

1.1.3.2 Operating System Noise

OS noise refers to the variations in application performance that are caused by the underlying operating system and hardware (Morari, Gioiosa, Wisniewski, Cazorla & Valero, 2011). These variations are caused by a wide variety of factors, which are discussed in Chapter 3. The effects of OS noise manifest themselves in measurements of an application's performance. Fine-grained performance measurements, known as

micro-benchmarks, demonstrate the effects of OS noise most clearly (Beckman, Iskra, Yoshii, Coghlan & Nataraj, 2008; M. Sottile & Minnich, 2004). Accordingly, any measurements of application performance that are recorded by the profiler of a computational offloading framework must be processed to prevent the presence of OS noise from interfering with the accuracy of the performance profile.

Most existing studies of OS noise have been performed on traditional desktop and server architectures (De, Kothari & Mann, 2007; Morari et al., 2011; Petrini, Kerbyson & Pakin, 2003). To date I have not found any existing research that performs a thorough study of OS noise on mobile device platforms. Existing approaches to mitigating the effects of OS noise on micro-benchmarking data are currently based on the characteristics of OS noise on traditional platforms. Before an approach to mitigating the effects of OS noise on micro-benchmarking data for mobile devices can be explored, data needs to be collected on the characteristics of OS noise found on mobile platforms.

1.2 Research Gaps, Questions, and Objectives

Despite the large number of existing computational offloading frameworks, the treatment of input data characteristics amongst the profiling and prediction components across the majority of these frameworks is still relatively simple. In addition, a thorough exploration of the issues surrounding fine-grained performance measurements on mobile platforms is largely lacking. Based on these limitations, I have identified the following gaps in the literature:

1. The profiling and prediction components of existing computational offloading frameworks do not comprehensively take into account the influence of arbitrary input data characteristics on application behaviour.
2. There has not been a thorough investigation into the levels of OS noise present on consumer mobile devices. In particular:
   (a) It is unknown if the levels and characteristics of OS noise on mobile platforms are consistent with observations of OS noise on traditional desktop devices and server architectures.
   (b) There has been no detailed investigation into the interactions between the changes in steady-state characteristic of mobile devices, and levels of OS noise.
3. Very few techniques for mitigating the effects of OS noise on micro-benchmarking data have been explored. Additionally, there has been no focus on optimising the efficiency of these techniques for use on resource-constrained mobile devices.

From these gaps in the literature, I have developed my research question:

- What is a feasible approach to building a profiling and prediction system for computational offloading that takes into account the influence of both arbitrary input characteristics and OS noise on application behaviour?

To answer this question, a series of smaller questions must first be addressed:

1. What levels of OS noise are present on consumer mobile devices?
2. Are the characteristics of OS noise present in micro-benchmarking data from consumer mobile devices consistent with those observed on other architectures?
3. Are OS noise levels influenced by changes in steady-state on mobile devices?
4. How can the effects of OS noise on micro-benchmarking results be mitigated in a manner that is both automated and efficient enough to be performed repeatedly on resource-constrained mobile devices?
5. What approach will allow a profiling and prediction system for computational offloading to be built that takes into account the influence of arbitrary input characteristics on application behaviour?
6. What approach will allow a profiling and prediction system that takes into account application input characteristics to be implemented in a manner efficient enough for use on resource-constrained mobile devices?

In order to address these questions, I define the following objectives for this study:

1. To measure the levels of OS noise present on consumer mobile devices;
2. To identify the characteristics of noisy micro-benchmark data measured on consumer mobile devices;
3. To establish if a relationship exists between OS noise levels and mobile device state changes;
4. To develop an automated technique for mitigating the effects of OS noise on micro-benchmarking results that is efficient enough for repeated use on resource-constrained mobile devices; and
5. To develop a profiling and prediction component, usable by a computational offloading framework, that accounts for the influence of arbitrary input characteristics on application behaviour, and is efficient enough for use on resource-constrained mobile devices.

The completion of each objective represents an answer to each of the associated research questions. Details on the solution to each objective are summarised in Section 1.4, described in the bridging statements at the start of Chapters 3 through 7, and summarised again in Chapter 8.

1.3 Research methodology overview

The research methodology utilised by the research in this dissertation is design science. Design science is a relatively young research methodology, having originated from the Information Systems discipline in the early 1990s (Peffers et al., 2007). Design science is focussed on problem solving (Hevner, March, Park & Ram, 2004). Instead of formulating one or more hypotheses and evaluating them, design science research identifies a problem to be solved and then proposes and evaluates a solution to that problem.

The research outputs of design science are referred to as *artifacts*. Artifacts contribute to the greater knowledge base and can take many forms, but are commonly prescriptive IT constructs (Gregor & Hevner, 2013). The Design Science Research Methodology (DSRM) is comprised of six activities for the development and refinement of artifacts. These activities are depicted in Figure 1.3. DSRM extends iterative models of software development and applies them to research. This iterative nature is reflected in a series of feedback loops that link the activities of the DSRM (Peffers et al., 2007).

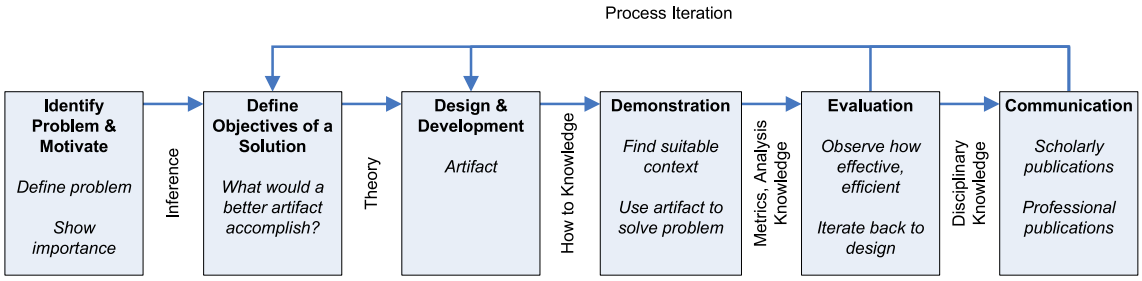


Figure 1.3: Design Science Research Methodology (DSRM) Process Model, adapted from Peffers et al. (2007).

The iterative nature of design science makes it particularly well-suited to research in technologically-focussed disciplines such as engineering, computer science, and information systems. The feedback loops connecting the activities of the DSRM facilitate rapid refinement of artifacts, resulting in more mature research contributions. Such refinement is key to achieving the levels of efficiency required by several of the research objectives listed in Section 1.2, particularly objectives 4 and 5. Due to these benefits, design science was selected as the ideal research methodology for this dissertation.

The overview of design science presented in this section includes only the details relevant to the rationale for selecting design science for use in this research. A more detailed discussion of design science and the activities of the DSRM is provided in Chapter 2.

1.4 Thesis Outline

This dissertation was undertaken as a thesis by publication. The thesis is divided into eight chapters. The introductory and concluding chapters are unique to the thesis itself and do not have any associated publications. Chapters 2 through 7 contain content that has been either published, submitted for publication, or is planned for submission, in the form of four journal articles and three conference papers, all double-blind peer reviewed.

Each chapter is prefaced by a brief abstract that discusses the research outcomes of the chapter, and lists the publications in which the chapter content has been disseminated. The chapters are organised as follows:

- Chapter 2 describes in detail the Design Science Research Methodology (DSRM) utilised by the research in this thesis, with a focus on evaluation methodology. Design Science is a popular research methodology that continues to evolve and mature. The evaluation activity of the DSRM draws on

evaluation methodologies from existing research disciplines. The application of these methodologies is guided by frameworks that consider the general characteristics of the research outputs, known as artifacts, that are being evaluated. Evaluation of artifacts for mobile device platforms presents a number of unique challenges that are not addressed by these existing frameworks. In this chapter, I address this gap by gathering and consolidating requirements from the existing literature on both evaluation methodology and mobile data collection. Based on these requirements, I propose a model for automating the evaluation of Design Science research artifacts targeting mobile platforms. The proposed model automates the deployment and execution of experiments on mobile devices, as well as the collection, transmission, and processing of data generated by experiments. A prototype implementation of the proposed model is utilised in the data collection of the subsequent data chapters of the thesis, with the exception of Chapter 5, which does not collect data from mobile devices.

- Chapter 3 addresses research objectives 1, 2, and 3, and addresses part of research objective 4. The two aims of this chapter are to examine the characteristics of OS noise on mobile devices, and to develop a technique to mitigate the effects of OS noise on micro-benchmarking results. To achieve this, I conduct a study of OS noise levels on Apple iPad Air devices. The study measures levels of OS noise present on devices whilst systematically varying the steady-state of each device. I examine the characteristics of the measured OS noise levels and compare them to the characteristics of OS noise on traditional desktop and server platforms reported by the existing literature. I then perform analysis to determine the relationship between OS noise levels and changes in system steady-state. After this, I propose an adaptive noise mitigation technique for use with micro-benchmarking data. The design of the noise mitigation technique is informed by the characteristics of the collected OS noise data. The initial version of the noise mitigation technique described in this chapter utilises a semi-automated outlier removal approach, which requires manual intervention to guide the adjustment of parameters. As a consequence, although the proposed noise mitigation technique addresses most of the requirements of research objective 4, the requirement for full automation is not yet satisfied at the conclusion of this chapter.

- Chapter 4 builds on the work presented in Chapter 3. The aim of this chapter is to develop a fully automated outlier removal approach for use with the adaptive noise mitigation technique proposed in Chapter 3. To improve the generality of the outlier removal approach, I first collect OS noise data from a variety of Apple iOS device models to supplement the existing data, which was only collected from iPad Air devices. I then examine the collected data to identify the nature of the outliers present, as well as any inherent clustering structures. Due to the nested clustering structure observed in the collected data, I select hierarchical clustering as the basis for my proposed outlier removal technique. From this starting point, I develop a heuristic for automatically determining the point at which to cut the dendrogram produced by hierarchical clustering, and remove clusters that are identified as outliers. I then simplify this heuristic to develop an approximation that features reduced computational complexity and is suitable for use on resource-constrained mobile devices. Finally, I validate the efficiency and effectiveness of the full and simplified versions of the proposed heuristic. Validation

demonstrates that the simplified heuristic produces results that are comparable to the full heuristic, at a significantly lower computational cost. This provides full automation of the noise mitigation technique proposed in Chapter 3, thus satisfying the associated requirements of research objective 4. Removal of outliers once hierarchical clustering has been completed is extremely efficient. However, the cost of performing hierarchical clustering itself is still too high for use on resource-constrained mobile devices.

- Chapter 5 extends the work presented in Chapter 4 by proposing a novel parallel algorithm for performing hierarchical clustering of single-dimensional data. Existing parallel implementations of hierarchical clustering do not maximise the level of parallelism. In particular, cluster merges are often performed in serial during part or all of the clustering process. My proposed parallel clustering algorithm makes use of General-Purpose Graphics Processing Unit (GPGPU) technologies to achieve massive parallelism on commodity consumer hardware. In addition, the unique characteristics of single-dimensional data are exploited to maximise the number of cluster merges that can be performed in parallel. This allows the algorithm to maximise the gains that are achieved from parallelism and improve efficiency. I validate the efficiency and correctness of my proposed algorithm against the classical serial algorithm. Validation results demonstrate that the clustering results produced by my proposed algorithm are equivalent to the results produced by the classical algorithm. Performance benchmarking demonstrates that my parallel algorithm achieves a significant speed gain when compared to the classical serial implementation. On graphics hardware supporting a sufficient number of simultaneous threads, the computational cost of my proposed algorithm is low enough to make it suitable for use on resource-constrained mobile devices. This satisfies the efficiency requirements of research objective 4.

- Chapter 6 builds the foundation for addressing research objective 5. This chapter aims to develop a model of application performance that takes into account the influence of arbitrary input characteristics, whilst remaining efficient enough for use on resource-constrained mobile devices. Existing software timing models are too computationally expensive for use on mobile devices and are not designed to adapt to changing device steady-states. After identifying timing models that are suitable for simplification, I select a timing model known as the *count-and-weights* model for use as the basis of my proposed model. I simplify the count-and-weights model to reduce information requirements and computational complexity, and also incorporate elements of a timing model known as the *pipeline* model to facilitate adaptation to changing device steady-states. I then validate the accuracy of the proposed timing model as an approximation of the count-and-weights model. Using both synthetic datasets and benchmarking data collected from a variety of embedded and mobile devices, I demonstrate that the proposed model is an extremely accurate approximation of the count-and-weights model for individual code execution paths. However, this chapter does not validate the predictive power of the proposed model for code with multiple execution paths.

- Chapter 7 completes the work that was started in Chapter 6 to address research objective 5. In this chapter, I propose a language- and platform-agnostic tooling pipeline that can be used to implement a

profiling and prediction system based on my proposed input-centric application performance model. Using this tooling pipeline, I implement a prototype of my proposed model that targets the C++ programming language and the Apple iOS platform. I then design an experiment to validate the predictive power of my proposed model for code with multiple execution paths. The experiment performs leave-one-out cross validation to demonstrate the ability of the proposed model to generalise to independent datasets. I also propose a heuristic to quantify the suitability of a given piece of code to prediction with the proposed model. I run the experiment on a set of Apple iPad Air devices using a series of code modules representing varying levels of the suitability heuristic. Results demonstrate that predictions produced by the proposed model are extremely accurate, even as the value of the suitability heuristic becomes less desirable. The proposed input-centric application performance model, and associated profiling and prediction prototype, provides extremely accurate predictions whilst remaining efficient enough for use on resource-constrained mobile devices. This satisfies the requirements of research objective 5.

The following table provides a breakdown of the intellectual contributions made by myself and others to the research in each chapter:

Chapter	Publications	Intellectual contributions
2	Rehn, A., Holdsworth, J., & Hamilton, J. (2014). A model for an automated evaluation framework for design science artifacts targeting mobile platforms. *Journal of Management Systems*, 24, 1–21.	Hamilton provided guidance on the nuances of the DSRM. Rehn conducted the literature review, developed the model, implemented the software prototype of the model, evaluated the effectiveness of the model, and wrote the paper content. Holdsworth and Hamilton provided editorial support.
3	Rehn, A., Hamilton, J., & Holdsworth, J. (2014). Towards an adaptive OS noise mitigation technique for microbenchmarking on mobile platforms. In *Fourteenth international conference on electronic business and the first global conference on internet and information systems* (p. 263-269). Rehn, A., Holdsworth, J., & Hamilton, J. (2014). Adaptive OS noise mitigation for microbenchmarking on mobile platforms. *Journal of Management Systems*, 24, 1–17.	Rehn performed the literature review, designed the experiment, performed the data collection, performed the data analysis, developed the adaptive noise mitigation technique, developed the semi-automated outlier removal approach, evaluated the noise mitigation technique, and wrote the paper content for both papers. Hamilton provided statistical support for the data analysis. Holdsworth and Hamilton provided editorial support for both papers.
4	Rehn, A., Holdsworth, J., & Lee, I. (2015). Automated outlier removal for mobile microbenchmarking datasets. In *10th international conference on intelligent systems and knowledge engineering (ISKE)* (p. 578-585).	Lee provided guidance on clustering evaluation metrics. Rehn performed the data collection, performed the data analysis, developed the automated outlier removal approach, conducted the analysis to develop the simplified heuristic for resource-constrained mobile devices, evaluated the two heuristics, and wrote the paper content. Rehn and Holdsworth wrote the algorithm pseudocode for the paper. Holdsworth and Lee provided editorial support.
5	Rehn, A., Possemiers, A., & Holdsworth, J. Efficient hierarchical clustering for single-dimensional data using CUDA. Submitted to *2017 Pacific Asia Conference on Information Systems (PACIS)*.	Possemiers provided guidance on the nuances of the CUDA API. Rehn developed the parallel clustering algorithm and the software implementation of the algorithm, with additional support from Possemiers. Rehn performed the literature review, designed the validation experiment, validated the parallel clustering algorithm, and wrote the paper content. Holdsworth provided mathematical and editorial support.

6	Rehn, A., Holdsworth, J., Hamilton, J., & Tee, S. An input-centric performance model for mobile applications. Submitted to *Journal of Systems and Software*.	Rehn developed the proposed timing model, with additional support from Holdsworth, Hamilton, and Tee. Rehn performed the literature review and identified the timing models suitable for simplification, designed the validation experiment, performed the benchmarking data collection and generated the synthetic datasets, validated the proposed timing model, and wrote the paper content. Holdsworth, Hamilton, and Tee provided editorial support.
7	Rehn, A., Holdsworth, J., Hamilton, J., & Tee, S. Input-centric performance prediction for mobile applications. To be submitted to *Journal of Systems and Software*.	Rehn developed the tooling pipeline to implement the proposed profiling and prediction approach, implemented the software prototype, designed the validation experiment, performed the data collection, validated the profiling and prediction approach, and wrote the paper content. Hamilton and Tee provided statistical support. Holdsworth, Hamilton, and Tee provided editorial support.

Chapter 2

Research methodology

This chapter discusses the Design Science Research Methodology (DSRM), with a focus on the evaluation of research outputs that target mobile device platforms. Drawing requirements from existing literature on both evaluation methodology and mobile data collection, I develop a model for the automated evaluation of design science artifacts targeting mobile platforms. The proposed model automates several feedback loops of the DSRM, facilitating rapid iteration and providing richer information to the researcher.

The prototype implementation of the model proposed in this chapter was utilised in the data collection for the subsequent chapters of the thesis. Without it, the collection and processing of data from such a large number and variety of mobile devices simply would not have been feasible. The only chapter to feature data collection that does not utilise the data collection framework is Chapter 5, which does not collect data from mobile devices.

The content from this chapter has been published as:

- *Rehn, A., Holdsworth, J., & Hamilton, J. (2014). A model for an automated evaluation framework for design science artifacts targeting mobile platforms.* Journal of Management Systems*, 24, 1–21.*

2.1 Introduction

Mobile device platforms have experienced significant growth in recent years. Modern, application-oriented mobile devices such as smartphones are replacing older fixed-function devices. For example, in 2013, sales of smartphones exceeded sales of feature phones for the first time (Gartner, Inc., 2014), reaching one billion shipments worldwide (International Data Corporation, 2014b). Modern mobile devices such as smartphones support sophisticated processing and customisation through the use of applications. Sales of applications for

mobile platforms continue to rise, in some areas exceeding sales of comparable software for other platforms (International Data Corporation, 2013a).

As mobile platforms continue to grow in popularity, focus has turned to the development of mobile applications (International Data Corporation, 2013b, 2014a). The timely development of robust products for mobile platforms presents a number of challenges, particularly with regards to evaluation and testing (Agarwal, Mahajan, Zheng & Bahl, 2010). Iterative software development methodologies facilitate rapid mobile application development, reducing time to market (Scharff & Verma, 2010). Researchers developing innovative mobile applications can benefit from a research paradigm that utilises an iterative methodology (Igler, 2013). Design science is a research paradigm whose iterative methodology facilitates rapid development and refinement of software products (Gregor & Hevner, 2013), making it well-suited to the development of mobile applications (Igler, 2013).

Design science gained popularity in the Information Systems discipline in the early 1990s (Peffers et al., 2007). Despite its relative youth, design science continues to gain acceptance as a legitimate research paradigm (Gregor & Hevner, 2013). The design science research methodology, initially explored by Hevner et al. (2004) and formalised by Peffers et al. (2007), continues to be analysed and refined as the paradigm matures. One aspect of the methodology that has gained attention in recent years is the evaluation of design science research outputs, termed *artifacts* (Gregor & Hevner, 2013; Peffers, Rothenberger, Tuunanen & Vaezi, 2012).

Existing artifact evaluation methodologies are drawn from mature research disciplines (Cleven et al., 2009). Guidance for applying evaluation methodologies is provided by general frameworks that consider artifact characteristics from a high-level perspective (Cleven et al., 2009; Venable, Pries-Heje & Baskerville, 2012). Existing frameworks do not provide specialisation of evaluation concepts to facilitate consideration of specific or technical artifact aspects. It is this gap that this chapter aims to address.

In this chapter I extend existing design science research evaluation methodologies by providing specialised consideration for the characteristics of artifacts targeting mobile platforms such as smartphones. Evaluating artifacts for mobile devices presents a number of unique challenges regarding data collection (Batalas & Markopoulos, 2012). By examining existing mobile data collection systems, I formulate a set of requirements for mobile artifact evaluation. I then present a model for an automated evaluation framework for evaluating design science artifacts targeting mobile platforms.

My proposed model extends existing design science research evaluation frameworks by automating data collection and processing, whilst allowing researchers to utilise the evaluation methodology of their choice. The automated collection and processing of data eliminates the bottlenecks of a manual workflow and allows researchers to focus on producing more refined artifacts.

The contributions of this chapter are as follows:

- I develop a model for an automated artifact evaluation framework for design science artifacts targeting mobile platforms. To my knowledge, no existing work applies automation to design science in the context of mobile devices. The framework provides an automated data collection and processing

workflow that satisfies requirements arising from characteristics common to artifacts targeting mobile devices. The workflow automates feedback loops of the DSRM. The workflow is extensible and customisable to facilitate requirements arising from characteristics not under consideration.

- I perform an initial validation of my model to demonstrate its potential. I compare the automated evaluation workflow facilitated by my framework to a manual workflow and find that the automated workflow offers numerous benefits to design science researchers.
- I frame the benefits of my model in context for both researchers and managers.

The rest of this chapter is structured as follows. First, I provide background on design science research methodology. I then examine the existing literature on artifact evaluation methodologies and mobile data collection frameworks. From the existing body of knowledge I draw requirements and formulate design goals for evaluation of artifacts designed for modern mobile devices. Next, I present my model for an automated evaluation framework for design science artifacts targeting mobile platforms. I then validate my proposed model and discuss the benefits it offers. I conclude by discussing the implications arising from this study, and explore potential directions for future work.

2.2 Background

2.2.1 Design science research methodology

Design science is a research paradigm with roots in engineering, and is identified as a distinct approach from traditional paradigms such as behavioural science (Gregor & Hevner, 2013). The primary focus of design science is problem solving (Hevner et al., 2004).

The focus of research paradigms influence their interactions with the knowledge base, which can be delimited into two distinct types of knowledge. The first type is "descriptive" knowledge, which describes laws that govern natural phenomena and relationships between them (Gregor & Hevner, 2013). The second type is "prescriptive" knowledge, which encompasses knowledge about *how* to do things, and human-created technological artifacts.

2.2.1.1 Artifacts

Artifacts are research outputs that contribute to the knowledge base and progress the understanding of problems being solved to a more mature state. Artifacts can include constructs (core concepts and symbols), models (abstract representations), methods (algorithms and techniques) and instantiations (implemented systems and products), as well as higher-level design theories regarding embedded phenomena (Hevner et al., 2004; Peffers et al., 2007). Research outputs of behavioural science are often descriptive knowledge, whereas research outputs of design science are often prescriptive IT artifacts. Development of these artifacts is informed by both the descriptive and prescriptive knowledge bases (Gregor & Hevner, 2013).

Design science differs from paradigms such as behavioural science by its methodology. The methodology of paradigms such as behavioural science begins with the formulation of one or more hypotheses. In contrast, design science research is motivated by the identification of a problem to be solved. A solution to the selected problem is then designed, and this solution is evaluated in place of a hypothesis (Peffers et al., 2007). Behavioural science research sees acceptance or rejection of a hypothesis resulting in a new, or interesting, contribution to the knowledge base. In contrast, design science research aims to identify problems to which solutions are believed to represent a valuable and useful contribution (Gregor & Hevner, 2013). For both paradigms, subsequent steps in each methodology are concerned with either the evaluation of the hypothesis or solution design.

DSRM comprises a number of activities. These activities are depicted in Figure 1.3.

2.2.1.2 DSRM Activities

The first and second activities of Figure 1.3 involve identifying of the problem and defining the objectives of a solution to the identified problem. These two activities encompass the review of the existing literature in search of significant gaps in existing knowledge. The second activity may be used as an alternate starting point if the research is motivated by a specific research or industry need (Peffers et al., 2007). As part of the iterative cycle, the second activity may be repeated as research objectives are revised and adapted, in light of new information gathered in subsequent activities.

The third activity is the design and development of the artifact that embodies the solution to the problem identified in activity one. Although other discoveries made during the research process may constitute additional artifacts, the artifact that embodies the designed solution is the focus of the methodology (Gregor & Hevner, 2013). The design and development activity is repeated a number of times to refine the artifact design during the development process, based on the results of subsequent demonstration and evaluation.

The fourth activity is demonstration of the artifact by applying it to an instance of the problem it aims to solve (Peffers et al., 2007). In the context of an artifact that represents a model or method, this activity typically involves the development of a prototype software implementation. It is this situated instantiation of the artifact that is evaluated in the subsequent activity.

The fifth activity is evaluation of the situated artifact, commonly through experimentation or simulation (Gregor & Hevner, 2013). The artifact is evaluated based on metrics defined by the researcher. Insights gained from the results of evaluation can then be used to improve the artifact. Interesting or useful aspects of the results are communicated through scholarly or professional publications, which serves as the sixth and final activity.

The middle phases of the design science methodology can form a loop that facilitates iterative design, demonstration, and evaluation. At the end of activities five or six the methodology may loop back to the either the second or third activity (Peffers et al., 2007). This feedback loop facilitates repeated refinement of an artifact as it is evaluated.

2.2.2 Design Science Research Artifact Evaluation

Existing Design Science Research (DSR) literature acknowledges the evaluation of artifacts as a key component of design science research. The evaluation activity provides evidence that an artifact represents a valuable contribution to the knowledge base (Gregor & Hevner, 2013). Venable et al. (2012) describe scientifically rigorous artifact evaluation as "what puts the 'science' in 'design science"' (p. 3). Despite the importance of the evaluation activity, DSR literature in previous years offered little guidance to researchers on selecting appropriate evaluation methodologies (Peffers et al., 2012).

Work in recent years has focussed on identifying and classifying evaluation methodologies used in existing design science projects (Cleven et al., 2009; Peffers et al., 2012; Venable et al., 2012). Peffers et al. (2012) examine a number of DSR articles and develop a taxonomy of evaluation methodologies. Cleven et al. (2009) characterise existing evaluation methodologies and develop an initial framework drawn from the greater bodies of Information Systems (IS), Computer Science (CS), and business administration literature. Pries-Heje, Baskerville and Venable (2008) develop a similar framework, which is expanded into a comprehensive framework by Venable et al. (2012). The frameworks and taxonomies presented in the existing literature consider artifact evaluation from a number of perspectives.

The framework presented by Pries-Heje et al. (2008) divides evaluation methodologies into a matrix containing four quadrants. The first dimension of the matrix represents the point at which evaluation is undertaken, and delineates activities into two categories: *ex ante* evaluation and *ex post* evaluation. Ex ante evaluation refers to the evaluation of a design or artifact prior to its instantiation (Pries-Heje et al., 2008). Ex post evaluation encompasses the formative evaluation of an instantiated artifact during its development and refinement, and the summative evaluation of an artifact after its construction is completed (Pries-Heje et al., 2008). Formative evaluation may be repeated iteratively during artifact refinement, and may vary from iteration to iteration as the researchers' understanding of the problem domain evolves (Peffers et al., 2007).

The second dimension of the matrix in the Pries-Heje et al. (2008) framework represents the evaluation context, and differentiates between a naturalistic context and an artificial context. Naturalistic evaluation evaluates an artifact in the real environment it is designed to be utilised in, whereas artificial evaluation evaluates an artifact in a controlled but inherently "unreal" setting (Venable et al., 2012).

The framework presented by Cleven et al. (2009) draws concepts from existing research evaluation literature and characterises evaluation activities using twelve dimensions. These dimensions represent artifact and evaluation characteristics and methodological approaches, depicted in Table 2.1.

Table 2.1
DSR artifact evaluation dimensions, adapted from Cleven et al. (2009). Note: vertical lines within dimensions are alternative characteristics or methodological approaches.

Dimension	Alternative Characteristics or Methodological Approaches						
Approach	Qualitative				Quantitative		
Artifact Focus	Technical			Organisational			Strategic
Artifact Type	Construct		Model	Method		Instantiation	Theory
Epistemology	Positivism				Interpretivism		
Function	Knowledge function		Control function		Development function		Legitimisation function
Method	Action research	Case Study	Field Experiment	Formal proofs	Controlled experiment	Prototype	Survey
Object	Artifact				Artifact Construction		
Ontology	Realism				Nominalism		
Perspective	Economic		Deployment		Engineering		Epistemological
Position	Externally				Internally		
Reference Point	Artifact against research gap			Artifact against real world			Research gap against real world
Time	Ex ante				Ex Post		

Among these dimensions are the time dimension of Pries-Heje et al. (2008) (ex ante vs. ex post), and a "reference point" dimension, which describes the point of reference used to evaluate the results of DSR (Cleven et al., 2009). Three reference points are described: evaluation of an artifact against a research gap, evaluation of an artifact against the real world, and evaluation of a research gap against the real world (Cleven et al., 2009). Other dimensions of the framework classify the methodological approach of the evaluation activity, as well as the type of artifact being evaluated. Types of artifacts are those described by the DSRM (Peffers et al., 2007). Remaining dimensions refer to underlying philosophies of research approaches (Cleven et al., 2009). A discussion of such philosophies is outside the scope of this thesis.

The dimensions depicted in Table 2.1 demonstrate a number of potential complexities of artifact and evaluation methodology characteristics. Recent works focus on generalised frameworks that aim to accommodate these characteristics and provide guidance to researchers in selecting an evaluation methodology (Peffers et al., 2012; Venable et al., 2012).

Venable et al. (2012) expand the framework presented by Pries-Heje et al. (2008) and develop an evaluation design methodology to guide researchers in utilising the extended framework. The presented design methodology encompasses several steps for selecting an evaluation methodology. These steps include considering the characteristics of the artifact being evaluated, and then identifying and analysing the purpose and requirements of the artifact evaluation activity (Venable et al., 2012). Artifact characteristics include whether the artifact is a product, process, or theory, and whether the artifact is purely technical or socio-technical. Potential purposes for evaluation presented include the validation of an artifact against its design goals, comparison of an artifact to other existing artifacts, and validation or enhancement of underlying design theories (Venable et al., 2012).

The components identified in the evaluation design methodology of Venable et al. (2012) reflect several dimensions of the earlier framework presented by Cleven et al. (2009). Artifact characteristics are considered by both frameworks from a high-level perspective. This high-level view facilitates the application of existing works as general frameworks (Cleven et al., 2009). These general frameworks present potential for extension to consider more specific artifact characteristics.

The concepts presented by the above evaluation frameworks can be specialised for automating the evaluation of artifacts targeting mobile platforms. To perform specialisation for automation, artifact characteristics must be considered at a more specific level. These specific characteristics instruct the requirements that must be satisfied in order to facilitate automation capabilities. Such characteristics include the technical complexities regarding data collection presented by mobile devices (Batalas & Markopoulos, 2012). These technical complexities are addressed by existing work on mobile data collection (Hoseini-Tabatabaei, Gluhak & Tafazolli, 2013). In the section that follows, I examine the existing literature on mobile data collection techniques and frameworks.

2.2.3 Mobile data collection frameworks

Rapid advances in mobile technologies in recent years have made mobile devices such as smartphones a very attractive platform for data collection (Hoseini-Tabatabaei et al., 2013). Mobile data collection systems have been presented for collecting experimental results (Brouwers & Langendoen, 2012; Emoto, Ohdachi, Watanabe, Sudo & Nagayama, 2006), survey responses (Brunette et al., 2013), and data from phone sensors for the purpose of user context recognition (Falaki, Mahajan & Estrin, 2011; Froehlich, Chen, Consolvo, Harrison & Landay, 2007; Hoseini-Tabatabaei et al., 2013).

Many recent mobile data collection systems focus on *in-situ* data collection from mobile devices in the real world (Batalas & Markopoulos, 2012). In-situ data collection introduces a number of unique considerations not present in data collection from mobile devices in a controlled laboratory environment. (Froehlich et al., 2007). To accommodate the varying characteristics of the data being collected, existing mobile data collection systems utilise a number of approaches. These approaches, and a number of requirements supported by existing literature, are summarised in Table 2.2 at the end of this section.

Froehlich et al. (2007) present MyExperience, a software system for collecting user feedback from mobile devices such as PDAs and mobile phones. MyExperience collects automated sensor and context data in addition to subjective user feedback. Self-reported user feedback is collected through surveys displayed to the user on the device's screen (Froehlich et al., 2007). Collected data is stored locally on the device and synchronised to a remote server opportunistically when a network connection is available. Researchers can then retrieve the collected data from the server (Froehlich et al., 2007). Researchers are able to customise the behaviour of MyExperience using arbitrary programming code. Customisations can be deployed to in-situ devices over a network connection (Froehlich et al., 2007).

Unlike MyExperience, SystemSens (Falaki et al., 2011) is a passive data collection system that does not interact with the user. SystemSens collects automated sensor data in the background and pairs the collected data with identifying information about the mobile device (Falaki et al., 2011). SystemSens stores data locally on the device and only uploads collected data to a remote server when the device is charging. The server then processes and visualises the collected data (Falaki et al., 2011).

Brunette et al. (2013) describe OpenDataKit, a data collection platform for Android smartphones. OpenDataKit is designed to be modular and extensible, and supports both the collection of automated sensor and context

data as well as survey responses (Brunette et al., 2013). OpenDataKit provides tools for researchers and organisations to design survey forms and deploy them to mobile devices. Collected data is stored locally on the device and shared between devices using peer-to-peer networking (Brunette et al., 2013). Transmission of collected data to the remote server is initiated manually by the user of the device through a submission interface provided by the software. The server can then export the collected data in a number of standardised formats or publish it to online services such as Google Docs (Brunette et al., 2013).

Brouwers and Langendoen (2012) present Pogo, a software system for managing experiments on mobile devices. The Pogo middleware runs on both the mobile devices and the computer of the researcher, utilising a central server for coordinating communication. The researcher's computer deploys experiments to mobile devices and acts as a collection node for receiving and storing collected experiment data (Brouwers & Langendoen, 2012). Experiments are defined by the researcher and may access the mobile device's sensor data via the Pogo middleware. Collected data is stored locally on the device and transmitted to the collection node opportunistically when the mobile device's network connection is active. Collected data may undergo processing on the mobile device prior to transmission, in addition to further processing on the collection node prior to storage in a database (Brouwers & Langendoen, 2012).

Emoto et al. (2006) describe the Kaiseki Server, a system for the collection of experimental data from Large Helical Device (LHD) experiments. Kaiseki Server supports collection of data from a wide range of mobile sensors and data acquisition devices (Emoto et al., 2006), and so the authors focus primarily on the central server itself. The server accepts data in a number of device-specific formats and performs conversion into a single unified data format. Since the collected datasets may be extremely large, the Kaiseki Server stores raw datasets on a fileserver and associates them with records in a relational database (Emoto et al., 2006). The server acts as a central repository only, and does not perform analysis or visualisation. Instead, researchers retrieve data from the server for processing and visualisation by specialised client applications (Emoto et al., 2006).

Batalas and Markopoulos (2012) review prior data collection frameworks and present a model formalising the requirements for in-situ data collection platforms. These requirements are categorised into the following groups (Batalas & Markopoulos, 2012):

- requirements regarding participants;
- requirements regarding researchers;
- requirements regarding development; and
- requirements regarding centralised servers.

2.2.3.1 Data collection requirements

Requirements from the perspective of participants include availability of the system on multiple device platforms and system unobtrusiveness. Data collection systems should facilitate participation in experiments

across a range of devices, whilst remaining as transparent as possible to the user (Batalas & Markopoulos, 2012).

Requirements regarding researchers include the ability to deploy arbitrarily complex systems, and the ability to rapidly modify experiments after they have been deployed (Batalas & Markopoulos, 2012). Researchers should not be constrained in the description of their experiments by limitations imposed by the data collection system. This requirement is also considered by Brouwers and Langendoen (2012), who opt for a general programming language for experiment description over a domain-specific language. Researchers should be able to rapidly modify experiments after deployment, to correct errors or modify experiment parameters (Batalas & Markopoulos, 2012).

Requirements regarding development include isolation of system components, discouragement of domain-specific languages and frameworks, and the ability for the system to cater to the needs of multiple stakeholders (Batalas & Markopoulos, 2012). The components of the system should be designed as a set of isolated layers that insulate one another from the effects of modifications to their implementation. The system should encourage the use of widely established practices and existing knowledge, favouring standardised languages and frameworks over custom-created or niche technologies (Batalas & Markopoulos, 2012). The system should be flexible enough to accommodate the specialised requirements of multiple stakeholders. This flexibility may be offered via customisation options for system components, or support for extensions utilising arbitrary programming code (Batalas & Markopoulos, 2012).

Requirements regarding centralised servers include an interface for storing and retrieving data, and the ability to process and visualise data (Batalas & Markopoulos, 2012). The centralised server should provide an interface to mobile devices to facilitate the transmission and storage of collected data. An interface should also be provided to researchers that allows them to query the stored data and perform processing and visualisation (Batalas & Markopoulos, 2012).

Rawassizadeh, Anjomshoaa and Tomitsch (2011) discuss the considerations that need to be made when storing data collected from pervasive devices, such as smartphones, for long-term archival purposes. The authors identify the use of standardised, widely-supported data formats as a key component for ensuring long-term access (Rawassizadeh et al., 2011). This approach is echoed in the works of Emoto et al. (2006) and Brunette et al. (2013), where the central server of the data collection system performs conversion of collected data into common, standardised formats.

Rawassizadeh et al. (2011) also suggest the use of metadata for enhancing information retrieval. This technique is utilised by SystemSens (Falaki et al., 2011), which pairs collected data with identifying information about the mobile device itself.

A number of features are present among the approaches utilised by existing data collection frameworks. Table 2.2 summarises these features and compares their inclusion in the existing frameworks discussed above.

Table 2.2
Summary of mobile data collection features.

Feature	Supporting Literature
Automated data collection and processing Data is collected without mobile device user intervention Collected data is transmitted without mobile device user intervention Collected data is processed without researcher intervention	Batalas and Markopoulos (2012); Brouwers and Langendoen (2012); Falaki et al. (2011); Froehlich et al. (2007)
Delayed data transmission Collected data is stored locally on mobile device Collected data transmission can be triggered at any point after collection	Brouwers and Langendoen (2012); Brunette et al. (2013); Falaki et al. (2011); Froehlich et al. (2007); Rawassizadeh et al. (2011)
Centrally stored data Collected data is stored on a central server Collected data is stored in a standardized format Collected data can be retrieved in a number of formats	Batalas and Markopoulos (2012); Brunette et al. (2013); Emoto et al. (2006); Falaki et al. (2011); Froehlich et al. (2007); Rawassizadeh et al. (2011)
Use of metadata Collected data is paired with collection metadata Device metadata is collected automatically Researchers can query collected datasets using metadata fields	Emoto et al. (2006); Falaki et al. (2011)
Modularity and extensibility Data collection code can be implemented in an arbitrary programming language Data processing and visualisation can be implemented in an arbitrary programming language	Batalas and Markopoulos (2012); Brouwers and Langendoen (2012); Brunette et al. (2013); Froehlich et al. (2007)
In-situ deployment Data collection code can be deployed to mobile devices Deployed data collection code can be modified by researchers remotely	Batalas and Markopoulos (2012); Brouwers and Langendoen (2012); Brunette et al. (2013); Froehlich et al. (2007)

The features of data collection frameworks can be adapted for use in automating the evaluation of design science research artifacts targeting mobile platforms. In the following section, I present my proposed model for automated artifact evaluation, based on the features identified above.

2.3 Model

In the sections that follow I present a model for an automated artifact evaluation framework, designed for the evaluation of design science research artifacts targeting mobile systems. First, I motivate my proposed model and discuss the design goals that guided its development. I then describe the architecture of the model in detail.

2.3.1 Objectives and Design Goals

The development of my proposed model was motivated by potential benefits of incorporating mobile data collection techniques into design science research artifact evaluation. In order to accommodate a range of existing evaluation methodologies and contexts, my design process focussed on two core objectives:

- **Facilitate both formative and summative evaluation.** Although summative evaluation of an artifact is often only performed once (Venable et al., 2012), formative evaluation may be performed repeatedly during the iterative artifact refinement phase. The evaluation methodology itself may also be refined as the researchers' understanding of the problem domain changes (Peffers et al., 2007). The model must allow researchers to rapidly modify and repeat the evaluation of an artifact as it is refined. The model must also facilitate easy comparison of results from previous evaluations to subsequent evaluation results, except in cases where modifications to the evaluation methodology render such a comparison of no value.

- **Facilitate both naturalistic and artificial evaluation.** An artifact may be evaluated in both a laboratory-based setting and its intended real-world environment over the course of its development (Venable et al., 2012). The model must support the requirements associated with in-situ data collection. However, providing such support must not introduce additional complexity for researchers when performing artificial evaluation.

To achieve these objectives, the following design goals were formulated based on requirements explored in the existing literature:

- **Automate data collection and processing.** The collection and processing of data must be an automated workflow. Researchers must be able to initiate an artifact assessment and view the processed and visualised results without requiring intermediate intervention (Brouwers & Langendoen, 2012; Falaki et al., 2011). An artifact assessment is the method used to evaluate the artifact, such as an experiment or survey.

- **Support delayed data transmission.** Collected data must be stored locally on the mobile device to permit transmission at a later time. This facilitates opportunistic synchronisation that may minimise overheads when evaluation is performed in-situ (Brouwers & Langendoen, 2012; Falaki et al., 2011; Froehlich et al., 2007).

- **Store data centrally in a standardised format.** Collected data must be stored on a central server. The server must transform data into standardised formats to facilitate interoperability and long-term archival (Brunette et al., 2013; Emoto et al., 2006; Rawassizadeh et al., 2011).

- **Use metadata for storing and querying data.** Collected data must be paired with metadata describing the artifact assessment parameters and mobile device characteristics. The metadata pertaining to device characteristics must be gathered from the mobile device automatically (Falaki et al., 2011). Metadata must be stored in a database and associated with the corresponding dataset (Emoto et al., 2006). Researchers must be able to query collected datasets using metadata fields.

- **Be modular and extensible in design.** The model must permit researchers to implement artifact assessments and data processing and visualisation using arbitrary tools and languages. The artifact assessment-specific and artifact-specific behaviour of the model must be customisable and extensible without limitation (Batalas & Markopoulos, 2012).

- **Facilitate re-processing of existing datasets using refined analyses.** The model must provide researchers the ability to apply refined implementations of processing and visualisation components to previously collected datasets. Researchers may refine evaluation methods during the development of an artifact, due to an improved understanding of the problem domain (Peffers et al., 2007) or to correct errors or oversights (Batalas & Markopoulos, 2012). The ability to process and visualise previously collected datasets using refined implementations also facilitates simpler comparison of existing and newly collected datasets.

- **Separate artifact assessment deployment from data collection.** The model must provide researchers with tools for managing deployment of artifact assessments to in-situ mobile devices (Batalas & Markopoulos, 2012; Brouwers & Langendoen, 2012). Such tools may introduce significant complexity (Brouwers & Langendoen, 2012). As such, they need to be isolated from other components of the model. Interactions between isolated components must be tightly controlled to facilitate clear separation and enforce intended component roles (Batalas & Markopoulos, 2012).

These goals guided the design of the architecture of my proposed model, which is described in the section that follows.

2.3.2 Model Architecture

In this section I describe the architecture of my proposed model for automated evaluation of design science research artifacts targeting mobile platforms. The high-level design of my proposed model is designed around a traditional client-server architecture (Batalas & Markopoulos, 2012; Emoto et al., 2006; Falaki et al., 2011; Froehlich et al., 2007). The model is split into three high-level components:

1. a mobile device middleware;
2. a central server; and
3. an artifact assessment deployment system.

Figure 2.1 depicts the interactions between these components.

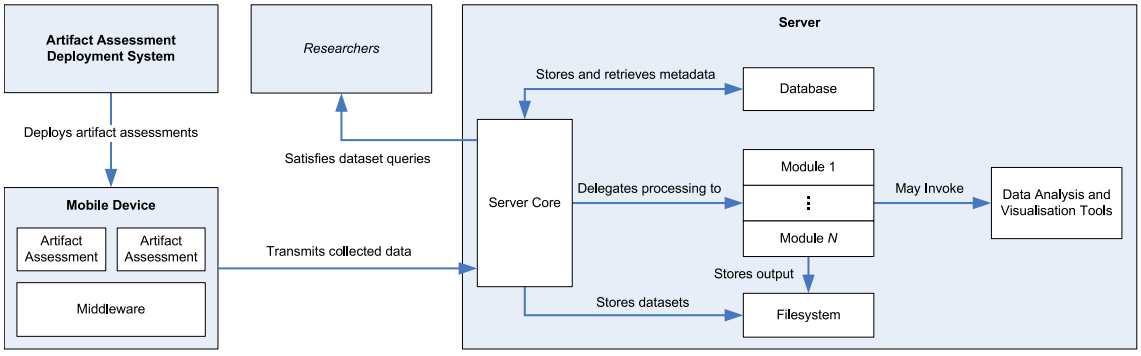


Figure 2.1: Interactions between the components of my proposed model for automated evaluation of design science research artifacts targeting mobile platforms.

The artifact assessment deployment system is used when performing in-situ evaluation, and deploys artifact assessment code to mobile devices. The mobile device middleware runs on mobile devices, coordinating the collection of data by artifact assessment code and managing the transmission of collected data to the server. The server acts as the central hub of the automated evaluation workflow. The server accepts collected data from mobile devices, coordinates the processing and storage of collected datasets, and services client requests for managing and querying stored datasets.

The mobile device middleware, server, and artifact assessment deployment system are all designed to satisfy the goals identified in the previous section. Each of these components are described in detail in the sections that follow.

2.3.2.1 Mobile Device Middleware

The mobile device middleware component runs on the mobile device. The middleware is responsible for interacting with the underlying device and for managing communication with the server. The details of this functionality is isolated to the middleware itself (Batalas & Markopoulos, 2012). Artifact assessments are specified using arbitrary programming code (Brouwers & Langendoen, 2012), and access device-specific functionality and server communication through the use of an Application Programming Interface (API) (Brouwers & Langendoen, 2012; Falaki et al., 2011). The middleware API allows artifact assessment code to query device characteristics, specify artifact assessment parameter metadata, and to initiate data transmission.

The middleware automatically detects device characteristics using device-specific functionality where needed. Characteristics include device vendor and model, operating system and version, as well as hardware information when available. When performing in-situ evaluation, information that uniquely identifies the device may also be gathered (Falaki et al., 2011). Detected device characteristics are made available to artifact assessments through the middleware API, and are used to populate metadata fields when performing data transmission.

In addition to the metadata gathered automatically by the middleware, metadata on artifact assessment parameters can be specified by artifact assessments through the middleware API. This metadata is specified for each dataset collected by an artifact assessment, and is stored alongside its corresponding collected data.

Artifact assessments store metadata and collected data on the local filesystem of the mobile device. Transmission of this data can then be triggered at a later point in time (Brouwers & Langendoen, 2012; Falaki et al., 2011; Froehlich et al., 2007). Artifact assessments can trigger data transmission explicitly through the middleware API. When performing in-situ evaluation, the middleware may automatically trigger data transmission, either periodically or opportunistically based on the device's state (Brouwers & Langendoen, 2012; Falaki et al., 2011; Froehlich et al., 2007).

Data transmission is performed using a protocol defined by the server. The server should use a standardised and widely supported protocol such as Representational State Transfer (REST), which is based on standard HTTP requests and responses (Fielding & Taylor, 2000). This allows the mobile device middleware to utilise existing HTTP protocol support present in modern mobile devices (Falaki et al., 2011; Froehlich et al.,

2007). Utilising existing device functionality reduces the complexity of the middleware implementation and maximises the use of existing tools and knowledge (Batalas & Markopoulos, 2012).

2.3.2.2 Server

The server acts as the central repository for collected data. The server is responsible for receiving, processing and storing data transmitted by the mobile device middleware. The server consists of several components that interact with one another to process data. The server components and their interactions are depicted in the lower half of Figure 2.1.

The server core is the component responsible for managing network communication and coordinating the other server components. The server core provides APIs that clients may access using one or more network protocols. One API facilitates receiving transmitted data and is utilised by the mobile device middleware for data transmission. Another API provides facilities for researchers to query and manage the collected data (Batalas & Markopoulos, 2012). Collected data can be queried and visualised based on metadata fields, and researchers can manually initiate re-processing of data that has been processed previously. A web interface is provided to facilitate easy access to the functionality of the researcher-oriented API (Batalas & Markopoulos, 2012; Falaki et al., 2011). The server core interacts with the other server components to satisfy client requests.

A database is utilised to store the metadata for collected artifact assessment datasets (Brouwers & Langendoen, 2012; Emoto et al., 2006; Falaki et al., 2011; Froehlich et al., 2007). Each entry contains both the automatically gathered metadata about the mobile device used to collect the data, as well as any metadata about artifact assessment parameters that was specified by the artifact assessment using the middleware API. The database does not contain the collected data itself (Emoto et al., 2006). Instead, each entry contains the location of the collected data on the filesystem.

All collected data is stored on the server's filesystem. The results of processing are also stored as separate files. Storing the results of processing as separate files makes processing non-destructive, as the original data is not modified. This allows collected data to be processed multiple times as evaluation methods are refined (Peffers et al., 2007). Processing is performed by modules coordinated by the server core.

The processing for each artifact assessment is specified using arbitrary programming code, in the same manner as the artifact assessment itself (Brouwers & Langendoen, 2012). Processing logic is bundled into modules. One module exists for each artifact assessment and contains the processing logic for that artifact assessment. When the server core satisfies a request to process a dataset collected by an artifact assessment, the corresponding module is located. Processing is delegated entirely to the module itself, allowing researchers to customise all aspects of processing to suit the needs of stakeholders (Batalas & Markopoulos, 2012). Modules may invoke existing data processing tools, maximising the use of existing tools and knowledge (Batalas & Markopoulos, 2012). Modules are managed using a plugin architecture (Wolfinger, 2008). This allows modules to be added, modified, and removed dynamically without the need to modify the server core.

2.3.2.3 Artifact Assessment Deployment System

The deployment system is a tool that provides researchers with facilities for managing in-situ artifact assessments deployed to real-world mobile devices. The deployment system maintains a pool of registered devices (Brouwers & Langendoen, 2012). Devices may belong to the researcher, or to volunteers who are participating in one or more artifact assessments (Batalas & Markopoulos, 2012). Each device may run multiple artifact assessments.

The deployment system provides an interface that allows researchers to manage the pool of registered devices, and manage the artifact assessments deployed to those devices (Batalas & Markopoulos, 2012; Brouwers & Langendoen, 2012). Researchers can add or remove devices. Researchers can deploy new artifact assessments to devices or revoke previously deployed artifact assessments. If one or more artifact assessments have been modified, researchers can deploy the modified version to devices that are running a previous version (Batalas & Markopoulos, 2012).

The deployment system interface may be implemented as either a web interface (Brunette et al., 2013), an API (Brouwers & Langendoen, 2012), or both. The deployment system and its interface should be isolated from the server components and their interfaces. This separation prevents the complexity of the deployment system from introducing complexity into server components, and enforces clear component roles (Batalas & Markopoulos, 2012).

In addition to the interface provided for use by researchers, the deployment system must communicate with mobile devices to facilitate the deployment of artifact assessments. The protocol utilised may be the same protocol used by the mobile device middleware to communicate with the server core. Use of the same protocol simplifies implementation and maximises the use of existing tools and knowledge (Batalas & Markopoulos, 2012).

The mechanism utilised by mobile devices for receiving artifact assessment deployments may vary based on device-specific considerations. On devices that support running tasks in the background, the mobile device middleware may manage deployed artifact assessments in the background (Brouwers & Langendoen, 2012). When deployed artifact assessments are managed in the background, the mobile device middleware provides an interface to device users to manage participation (Batalas & Markopoulos, 2012; Brouwers & Langendoen, 2012). The participation management interface allows users to opt in to, or out of, participation in individual deployed artifact assessments.

2.4 Model validation

To validate my proposed model, I implemented a prototype server and prototype mobile device middleware for the Apple iPhone platform. Due to technical limitations of the iPhone platform regarding security constraints, I did not implement a prototype of the evaluation deployment system.

Validation of my proposed model focussed on the formative evaluation of a technical artifact in an artificial environment. The artifact assessment took the form of a software simulation run on multiple iPhone devices.

Using the iterative design science research methodology of Peffers et al. (2007), I repeatedly evaluated and refined the artifact.

Several iterations were performed without the use of the prototype system. Data collected by the artifact assessment was manually retrieved from the mobile devices using the Apple Xcode development tool (Apple Inc., 2014), as depicted in Figure 2.2. Processing and visualisation was manually invoked for each collected dataset. Processed datasets were then stored on the local filesystem, manually placed within directories to represent metadata information. No database was used. Manual data retrieval and processing was a time-consuming task. Locating and comparing datasets was also a cumbersome process.

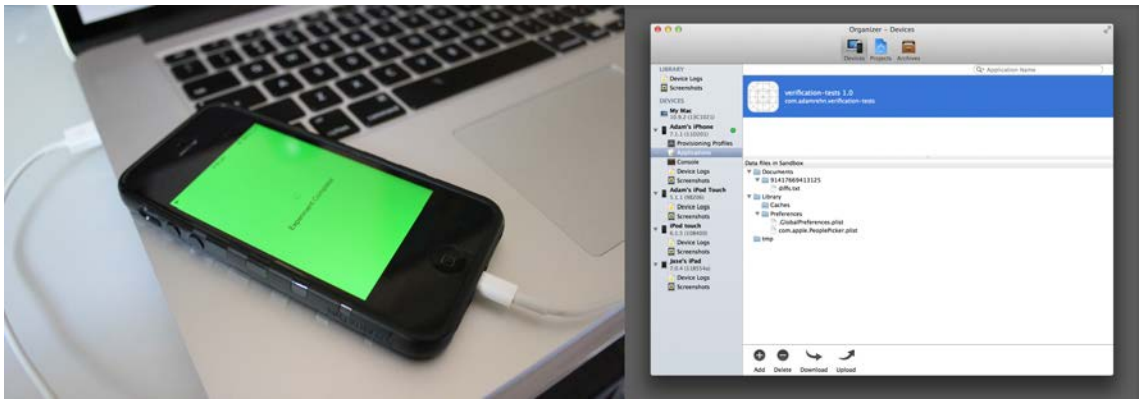


Figure 2.2: Manual data collection using the Apple Xcode development tool.

During the iterations performed without the prototype system, my understanding of the problem the artifact was designed to solve improved (Peffers et al., 2007). I modified the artifact assessment accordingly, including the processing and visualisation tools. The datasets from previous iterations were still directly comparable to datasets collected during subsequent iterations. In order to make a meaningful comparison, it was necessary to re-process all existing datasets using the modified processing and visualisation tools. This was an another time-consuming task. The large amount of time required for both manual data retrieval, initial data processing, and subsequent data re-processing precluded rapid iterations and evaluation refinements.

Several iterations were then performed using the prototype of my proposed model. During these iterations, I continued to refine the artifact assessment and modify the data processing and visualisation tools. Each time I modified the artifact assessment, all existing datasets were re-processed using the modified tools. By comparing the iterations performed without the prototype of my proposed models to the iterations performed with the prototype, I observed several advantages of my proposed model:

- *Elimination of bottlenecks.* Simultaneous automated data collection and processing for multiple devices was found to be significantly faster than performing the equivalent manual workflow for a single device. Automated re-processing of existing datasets was also found to be significantly faster than performing re-processing manually. These findings indicate that the time required for data transmission and processing is less significant than the time required for researchers to manually interact with software tools.

- *Ease of dataset comparison.* When collected datasets were stored without a database to associate them with metadata information, locating and comparing datasets was a slow and cumbersome process. Using the interface provided by the server core, it was fast and simple to query datasets based on metadata fields and compare datasets and visualisations. Although a database could be utilised without the rest of my proposed model, the automated workflow of my model alleviates the requirement for cumbersome manual data entry.
- *Improved information quality.* Due to the ease with which modified versions of the processing and visualisation tools could be applied to existing datasets, I was able to refine my processing methodology repeatedly. Processing and visualisation tools could be improved independently of data collection, and re-applied to either all existing datasets or a subset of them. This facilitated the ability to ask more questions of the collected datasets and extract more meaningful information.
- *Rapid iteration and refinement.* The iterations performed with the prototype of my proposed model were significantly faster than the iterations performed without the prototype. Rapid iteration times facilitated a greater number of total iterations, ultimately resulting in a more refined and mature artifact. The prototype of my proposed model allowed me to maximise the benefits inherent in the iterative nature of the DSRM of Peffers et al. (2007).

2.5 Implications of research

Although the initial validation of my proposed model used only a limited prototype, results demonstrate that my proposed model provides clear advantages for researchers evaluating DSR artifacts for mobile platforms. In this section I discuss the theoretical and practical implications arising from this study, as well as the implications for management.

2.5.1 Theoretical implications

Design science continues to undergo refinement as a research paradigm (Gregor & Hevner, 2013). The evaluation phase of the DSRM is analysed and refined in recent work, and presents further potential for significant extension and refinement (Cleven et al., 2009; Peffers et al., 2012; Pries-Heje et al., 2008; Venable et al., 2012). My proposed model represents one possible extension.

The development of my proposed model was guided by requirements drawn from the existing literature on mobile data collection. These requirement represent existing solutions to the problem of collecting data from mobile devices. This study focussed on adapting these existing solutions to the problem of evaluating DSR artifacts targeting mobile platforms. Gregor and Hevner (2013) term the extension of existing solutions to new problems *exaptation*. Exaptation is identified as a key research approach in their DSR knowledge contribution framework (Gregor & Hevner, 2013).

The methodologies commonly used in DSR artifact evaluation are drawn from existing research disciplines (Cleven et al., 2009; Venable et al., 2012). The benefits demonstrated by my proposed model highlight the

potential of further exaptation of existing practices for use in evaluating design science research artifacts. Potential candidates for exaptation include practices from both the research community and industry (Sedita, 2012).

The exaptation represented by my proposed model is specific to artifacts targeting mobile platforms. This study was motivated by the identification of commonality between technical characteristics of evaluating artifacts targeting mobile platforms and collecting data from mobile devices. The identification of connections between concepts in disparate fields or contexts is core to exaptation (Gregor & Hevner, 2013). Further exaptations may target other artifact characteristics. These characteristics are influenced by factors such as whether artifacts are products, processes, or theories, and whether artifacts are purely technical or socio-technical (Venable et al., 2012).

2.5.2 Practical and management implications

For researchers evaluating design science research artifacts targeting mobile platforms, the practical implications of this study are clear. My proposed model can be implemented and utilised to facilitate an automated artifact evaluation workflow. This workflow automates the feedback loop of the design science research methodology, as depicted in Figure 2.3.

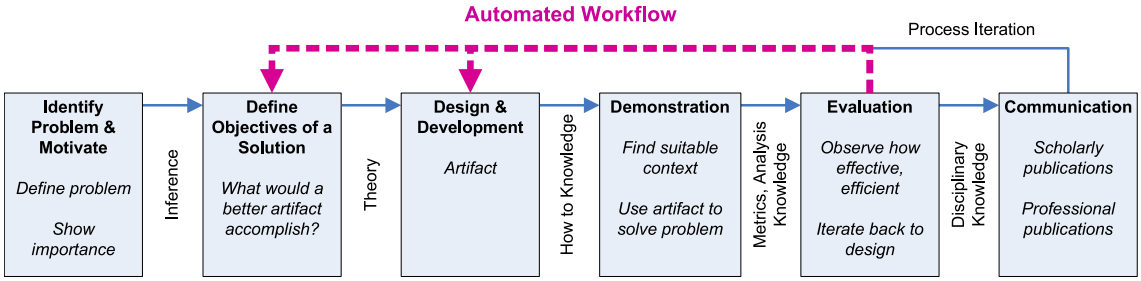


Figure 2.3: DSRM process iteration loops (Peffers et al., 2007) automated by the automated evaluation workflow.

The feedback loop from the evaluation activity to the design and development activity is automated by the components of the automated workflow. The feedback loop from the evaluation activity to the objective definition activity is automated by the facility to re-process previously collected data using refined analyses, as researchers' understanding of the problem changes. The feedback loop from the communication activity is not automated, as it is the responsibility of the researcher to consider the insights of reviewers and citing researchers.

The use of an automated artifact evaluation workflow provides a number of benefits, such as faster iteration times and improved information quality. (See the validation of my proposed model for a detailed discussion.) These benefits can lead to the production of more refined artifacts, resulting in more mature research contributions (Gregor & Hevner, 2013).

For managers, my proposed model offers several benefits:

- Improved information quality and the elimination of bottlenecks caused by manual data collection facilitate greater research productivity. Faster iterations can result in production of more refined artifacts and, ultimately, more mature products.
- Collected data is stored centrally and can be exported to a number of standard formats (Brunette et al., 2013). This facilitates the fulfilment of any data storage and management requirements the researchers or organisation must adhere to.
- Processing and visualisation of collected data can be delegated to existing data analysis tools. This maximises return on the investment such tools represent.

The extensibility and customisability of my proposed model offers further benefits for management. The automated workflow of my proposed model can be customised and extended to meet individual researchers' and stakeholders' needs (Batalas & Markopoulos, 2012). Specialised data analyses and visualisations can be produced for all relevant parties or departments. The automated workflow can be extended to facilitate automatic report generation and dissemination of processed data to interested parties.

2.6 Future Research

In future, I intend to implement all of the features of my proposed model and perform more detailed validation using multiple mobile device platforms and DSR artifacts. I intend to perform case studies involving DSR projects that incorporate both ex ante and post ante evaluation (Pries-Heje et al., 2008) as well as evaluation in both naturalistic and artificial environments (Venable et al., 2012).

Potential future extensions to this study include integration with e-Research data management tools. These tools operate in a similar manner to my proposed model, facilitating the collection, processing, and central storage of generated research data (Androulakis et al., 2009). However, e-Research focuses on a far larger scope than my work, incorporating the use of scalable grid computing resources and collaboration tools for researchers (Paterson, Lindsay, Monotti & Chin, 2007). Integration with the larger scope of e-Research presents interesting potential directions for future extension of the work presented in this chapter.

Potential exists for other research following the process utilised in the development of my proposed model. My proposed model adapts existing knowledge and practices for use in evaluating design science research artifacts. Existing knowledge and practices were selected by identifying commonalities between known problems and characteristics specific to artifacts targeting mobile platforms. There is potential for future work to focus on other artifact characteristics. By identifying commonalities between other characteristics and problems with known solutions, existing knowledge and practices can be adapted for use in evaluating artifacts featuring those characteristics. In this manner, evaluation of design science research artifacts can be further refined to accommodate specific requirements arising from the nature of artifacts.

2.7 Conclusion

Mobile device platforms continue to experience strong growth, driving the development of mobile applications. The development and evaluation of products for mobile platforms presents a number of unique challenges. Iterative development methodologies facilitate rapid development and refinement, reducing time to market. Researchers developing innovative mobile products can benefit from the iterative methodology of design science.

The evaluation activity of the DSRM continues to be formalised, and presents opportunities for refinement and extension. Existing generalised evaluation methodologies do not facilitate the unique technical challenges of automated evaluation of artifacts targeting mobile platforms. These technical challenges are addressed by existing work on mobile data collection systems. The techniques presented by these mobile data collection systems can be applied to the context of design science to facilitate automated artifact evaluation.

In this chapter, I presented a model for an automated evaluation framework for design science artifacts targeting mobile platforms. I drew requirements from the existing literature on both artifact evaluation and mobile data collection. From these requirements I formulated a set of design goals for my model. To satisfy these design goals, my proposed model facilitates an automated, extensible and customisable artifact evaluation workflow. The automated workflow supports both formative and summative artifact evaluation in naturalistic and artificial environments. The evaluation framework of my model utilises client-server communication to provide a central point of control for the processing and storage of collected data. A deployment system component provides researchers with tools to deploy and manage artifact assessments on mobile devices.

The automated evaluation workflow of my proposed model automates feedback loops of the design science research methodology. The automated workflow provides a number of benefits to researchers, including faster iterations and improved information quality. These benefits facilitate the production of more refined artifacts and consequently more mature research contributions. For managers, the extensibility and customisability of the automated workflow facilitates satisfaction of the needs of all stakeholders.

By utilising an automated evaluation workflow, researchers and developers can rapidly develop more robust products for mobile device platforms. Automated evaluation addresses the unique technical challenges presented by mobile device platforms and provides organisations a clear advantage in the growing and competitive application-oriented mobile marketplace.

Chapter 3

Operating System (OS) noise study

This chapter addresses research objectives 1, 2, and 3, and builds the foundation for addressing research objective 4. In this chapter, I conduct a study of OS noise on Apple iPad Air devices. OS noise levels are measured and characterised. Examination of the collected data confirms that the characteristics of OS noise on mobile devices are consistent with those observed on traditional desktop and server platforms in the existing literature. Analysis of the collected data demonstrates that OS noise does indeed interact with changes in device steady-state.

After analysing the collected data, I propose an adaptive noise mitigation technique for use with micro-benchmarking data. The proposed technique mitigates the effects of OS noise so that accurate and meaningful results can be obtained from micro-benchmarking data. This addresses all of the requirements of research objective 4, with the exception of being fully automated. The outlier removal process described in this chapter is only semi-automated. A fully automated outlier removal technique is subsequently developed in Chapter 4.

The content from this chapter has been published as:

- *Rehn, A., Hamilton, J., & Holdsworth, J. (2014). Towards an adaptive OS noise mitigation technique for micro-benchmarking on mobile platforms. In* Fourteenth international conference on electronic business and the first global conference on internet and information systems *(p. 263-269).*
- *Rehn, A., Holdsworth, J., & Hamilton, J. (2014). Adaptive OS noise mitigation for micro-benchmarking on mobile platforms.* Journal of Management Systems*, 24, 1–17.*

3.1 Introduction

Micro-benchmarking is a subset of performance benchmarking that examines the performance characteristics of individual parts of a software application, as opposed to the entire application (Staelin, 2005). Micro-benchmarking is an effective tool for analysing the performance of individual application components, such as user interface responsiveness (Endo, Wang, Chen & Seltzer, 1996). Responsiveness of interactive mobile applications has been demonstrated to significantly influence user perception (Tolia, Andersen & Satyanarayanan, 2006). Micro-benchmarking potentially represents an important tool in the optimisation of mobile device platforms and applications. However, in spite of the potential uses in a mobile context, micro-benchmarking receives relatively little attention compared to entire-application benchmarking. This is perhaps due to the limitations of micro-benchmarking approaches.

A significant limitation of micro-benchmarking, regardless of platform, is that it is susceptible to interference from the underlying operating system and hardware (S. Wang et al., 2002). Such interference is referred to as Operating System (OS) noise (Akkan, Lang & Liebrock, 2012), and results in performance variations that manifest in micro-benchmark measurements (S. Wang et al., 2002). I observe that existing micro-benchmarking implementations fail to provide a comprehensive approach to mitigate the effects of OS noise on micro-benchmark results.

In this chapter I study OS noise characteristics on real-world mobile devices, specifically exploring the relationships between device state and noise levels on Apple iPad devices. The primary motivation of this study is to determine those characteristics relevant in a micro-benchmarking context.

The contributions of this chapter are as follows:

- I perform a study to examine OS noise on Apple iPad devices. I am not aware of any previous studies that examine OS noise in a mobile context.
- I propose an adaptive noise mitigation technique for use in micro-benchmarking.
- I validate the proposed technique using data collected from the mobile device study.

The rest of this chapter is structured as follows. First, I explore the existing literature on OS noise in a number of contexts, and motivate the study by highlighting the existing knowledge gaps. I then describe the methodology and results of the mobile device OS noise study. Finally, I propose an adaptive noise mitigation technique, and evaluate it using data collected from the OS noise study.

3.2 Background

Operating System (OS) noise is the interference to application performance caused by the operating system and hardware (Seelam et al., 2013). Processor time spent performing background system activities results in variations in application performance (Petrini et al., 2003).

As there remains insubstantial existing work on OS noise in the context of mobile devices, I examine OS noise in the context of parallel computing, where it has been studied extensively (Akkan et al., 2012). I then explore existing work on OS noise in the context of micro-benchmark accuracy.

3.2.1 Impact of OS noise within parallel computing

Extensive research has been undertaken into the impacts of OS noise within parallel computing (Akkan et al., 2012). Many parallel computing applications utilise a bulk-synchronous computation model, whereby processing is performed in lock-step and nodes synchronise with one another at the completion of each step, before the next step can begin (Seelam et al., 2013). Synchronisation takes place only after all nodes have completed processing the step. As a result, delays in processing on even a single node, delay the completion of the step for all nodes (Seelam et al., 2013). The accumulation of these small delays over the course of a parallel application's execution can amplify their impact, resulting in severe performance degradation and reduced scalability (Garg & De, 2006).

In this section, I describe the sources of OS noise commonly identified in the existing literature and the techniques used to measure OS noise.

3.2.1.1 Sources of OS noise

OS noise can be caused by a large number of sources. A number of common sources of OS noise identified in the existing literature are listed in Table 3.1.

Table 3.1
Commonly identified sources of OS noise from the existing literature. Sources can arise from the underlying hardware and operating system, as well as services running on behalf of the operating system.

Noise source	Supporting Literature
Cache misses	De et al. (2007); Tsafrir (2007); Tsafrir, Etsion, Feitelson and Kirkpatrick (2005)
Context switches and pre-emptive scheduling	De et al. (2007); Nataraj, Morris, Malony, Sottile and Beckman (2007); Tsafrir (2007); Tsafrir et al. (2005)
Interrupts	Akkan et al. (2012); De et al. (2007); Ferreira, Bridges and Brightwell (2008); Morari et al. (2011); Petrini et al. (2003); Tsafrir (2007); Tsafrir et al. (2005)
OS clock ticks and timers	Akkan et al. (2012); Ferreira et al. (2008); Morari et al. (2011); Nataraj et al. (2007); Tsafrir (2007); Tsafrir et al. (2005); S. Wang et al. (2002)
Page faults	De et al. (2007); Morari et al. (2011)
System daemons	De et al. (2007); Ferreira et al. (2008); Morari et al. (2011); Petrini et al. (2003); Tsafrir (2007); Tsafrir et al. (2005)
Translation Lookaside Buffer (TLB) misses	De et al. (2007); Morari et al. (2011)

Petrini et al. (2003) investigate performance limitations of applications running on the ASCI Q supercomputer. The authors observe, and analyse, performance variations on a per-node basis, and then investigate the impact on overall application performance when these small delays are amplified due to the nature of bulk-synchronous computation. System daemons and interrupts generated by the cluster management software utilised are identified as key sources of noise (Petrini et al., 2003).

Operating system clock ticks are identified as a key source of noise by Tsafrir et al. (2005). Clock ticks are interrupts generated by the operating system at a regular fixed interval so that the operating system can perform runtime tasks and provide timing services to applications (Akkan et al., 2012; Tsafrir et al., 2005). Tsafrir et al. (2005) analyse OS noise on a number of platforms and system configurations, and propose a dynamic timer system to replace the use of periodic timers, in order to reduce the noise generated by clock ticks.

De et al. (2007) develop a framework that utilises kernel instrumentation to identify and characterise sources of OS noise. A large number of noise sources are identified, including context switches, daemons, and interrupts. The authors also identify cache misses, page faults, and translation lookaside buffer (TLB) misses as potential sources of noise, but leave collection of data on these factors as future work (De et al., 2007).

Tsafrir (2007) measures OS noise and analyses its impact on context switching overhead across several processor architectures. Cache misses caused by kernel code are identified as a large contributor to the measured noise (Tsafrir, 2007). Other sources including daemons, interrupts, and OS clock ticks are also identified as contributors to noise.

Nataraj et al. (2007) present extensions to the KTAU kernel instrumentation tool to integrate it with application-level performance monitoring tools, and utilise the system to characterise sources of OS noise. OS clock ticks and timers are identified as key sources of noise, as well as pre-emptive process scheduling mechanisms that utilise clock ticks (Nataraj et al., 2007).

Morari et al. (2011) present yet another framework that utilises kernel instrumentation to characterise the sources of OS noise. Numerous noise sources are identified, including daemons, interrupts, page faults, TLB misses, and OS clock ticks and timers (Morari et al., 2011).

Akkan et al. (2012) propose changes to the Linux kernel to minimise the level of OS noise. A soft-partitioning scheme is proposed, whereby operating system activities are isolated to a single processor core, whilst applications execute uninterrupted on the remaining cores (Akkan et al., 2012). The authors also implement a dynamic clock tick system, similar to that proposed by Tsafrir et al. (2005), which they refer to as a "tickless" kernel (Akkan et al., 2012).

The sources of OS noise are complex and interrelated, and cannot be trivially removed (Akkan et al., 2012). Elimination of the underlying sources of OS noise is beyond the scope of this study. However, the effects of OS noise can be reduced if OS noise is measured and then characterised.

3.2.1.2 Measuring OS noise

A number of common techniques for measuring OS noise are prominent in the existing literature. Details of these measurement techniques are listed in Table 3.2.

Table 3.2
Common techniques for measuring OS noise from the existing literature.

Measurement technique	Implementation	Supporting Literature
Fixed Work Quantum (FWQ) *(User-level micro-benchmark)*	Performs a fixed amount of processing in a loop, recording the system timestamp each time to determine the timestamp delta (execution time) of each iteration.	Akkan et al. (2012); Petrini et al. (2003); Tsafrir (2007); Tsafrir et al. (2005)
Fixed Time Quantum (FTQ) *(User-level micro-benchmark)*	Measures the amount of work performed in a fixed amount of time, using a nested loop to ensure each outer loop iteration takes a fixed time to run. The periodicity of the generated data makes it suitable for use with signal processing techniques.	M. Sottile and Minnich (2004); M. J. Sottile, Chandu and Bader (2006)
Selfish detour *(User-level micro-benchmark)*	Records timestamp deltas in a tight loop, storing only those deltas that are above a threshold. This threshold can be predefined or relative to some metric, such as the minimum observed delta.	Beckman, Iskra, Yoshii and Coghlan (2006); De et al. (2007); De and Mann (2010); De, Mann and Mittaly (2009); Ferreira et al. (2008); Hoefler, Schneider and Lumsdaine (2010); Nataraj et al. (2007); Seelam et al. (2013)
Kernel instrumentation + user-level micro-benchmark	Utilises kernel instrumentation to record operating system events, and makes this data available to a user-level benchmark that records noise using one of the techniques listed above. Operating system events are paired with collected noise profiles so noise can be correlated with the underlying source.	De et al. (2007); Nataraj et al. (2007)
Kernel instrumentation only	Removes the use of a user-level micro-benchmark entirely, generating noise profiles directly from data collected by kernel instrumentation.	Morari et al. (2011)

Most of the measurement techniques utilise a user-level micro-benchmark to measure performance variations caused by OS noise. These micro-benchmark techniques each build upon their predecessors, steadily increasing in complexity and sophistication.

Fixed Work Quantum (FWQ) is the oldest and simplest micro-benchmarking technique utilised in the existing literature on OS noise. FWQ is based on the principle of repeatedly performing a fixed quantity of processing and recording the time taken to complete each processing cycle (Akkan et al., 2012). Processing is performed in a small loop. The system timestamp is recorded at the beginning of each iteration, and after the loop completes. Each pair of timestamps is compared, then the system stores the timestamp deltas, which represent the time taken to perform each iteration (De et al., 2007). FWQ has the benefit of being extremely simple to implement. However, this benefit is weighed against two notable limitations.

First, because the time intervals being recorded by FWQ vary in length as a result of noise, the generated data

lacks periodicity. This precludes the use of signal processing techniques when analysing the data (M. Sottile & Minnich, 2004). Second, because all of the raw data generated by the micro-benchmark is stored in memory, larger iteration counts may influence the results by introducing noise caused by memory access effects such as cache misses (M. Sottile & Minnich, 2004). This precludes the use of FWQ for recording noise traces over a long period of time.

Fixed Time Quantum (FTQ) is designed to address the first limitation of FWQ (M. Sottile & Minnich, 2004). FTQ is effectively an inversion of FWQ, measuring work performed per unit of time, as opposed to time taken to perform a unit of work. A loop counts the amount of processing performed by the system, running until a fixed interval of time has elapsed (M. Sottile & Minnich, 2004). As a result, the generated data is periodic, and can be analysed using signal processing techniques. These techniques can be useful in relating recorded noise events to the underlying noise sources that cause them (M. Sottile & Minnich, 2004). Just as the sampling rate of FWQ is driven by the size of the amount of work being performed, the sampling rate of FTQ is controlled by the time quantum used (Hoefler et al., 2010).

Selfish detour addresses the second limitation of FWQ (Hoefler et al., 2010). Selfish detour is an extension of the fixed work principle, and records timestamps in a tight loop with no additional processing (Beckman et al., 2006). This minimises the amount of work being performed, thus maximising the sampling rate (Hoefler et al., 2010). In order to decrease the amount of raw data generated, timestamp deltas are only recorded if they fall above a threshold (Beckman et al., 2006). In the original implementation by Beckman et al. (2006), this threshold is predefined. In a later implementation by Hoefler et al. (2010), the threshold is dynamically determined, relative to the minimum observed value. This threshold filtering mechanism reduces the amount of data that needs to be stored in memory (Beckman et al., 2006), facilitating the recording of noise traces over a longer period of time than feasible when using FWQ.

In addition to the use of micro-benchmarks, several existing works utilise kernel instrumentation to measure OS noise and correlate noise events to underlying noise sources (De et al., 2007; Morari et al., 2011; Nataraj et al., 2007).

De et al. (2007) combine the use of micro-benchmarks and kernel instrumentation to gain greater insights into the characteristics of OS noise. Kernel instrumentation is utilised to record information about all kernel events. The recorded information, along with all internal kernel data structures, is then made available to the user-level micro-benchmark (De et al., 2007). The micro-benchmark is implemented using a version of the selfish detour technique. The recorded noise information can then be matched to kernel events recorded by the kernel instrumentation data (De et al., 2007).

Nataraj et al. (2007) utilise a similar approach, modifying the existing KTAU kernel instrumentation tool to make the data generated by it accessible by a user-level micro-benchmark. The micro-benchmark is based on the selfish detour technique (Nataraj et al., 2007).

Morari et al. (2011) remove the use of a micro-benchmark entirely, relying entirely on kernel instrumentation. The existing Linux Trace Toolkit Next Generation (LTTng) kernel instrumentation tool is extended to provide more detailed information on kernel events, including OS noise. Morari et al. (2011) compare the data

generated by the kernel instrumentation to that generated by FTQ, and find that it is of comparable accuracy, whilst containing more information about the sources of the OS noise.

Existing OS noise measurement techniques fall into two main approaches. These approaches, and their outputs, are depicted in Figure 3.1. As I find no supporting studies that replicate the collection of OS noise profiles through kernel instrumentation, as performed by Morari et al. (2011), this technique is excluded from the categorisation.

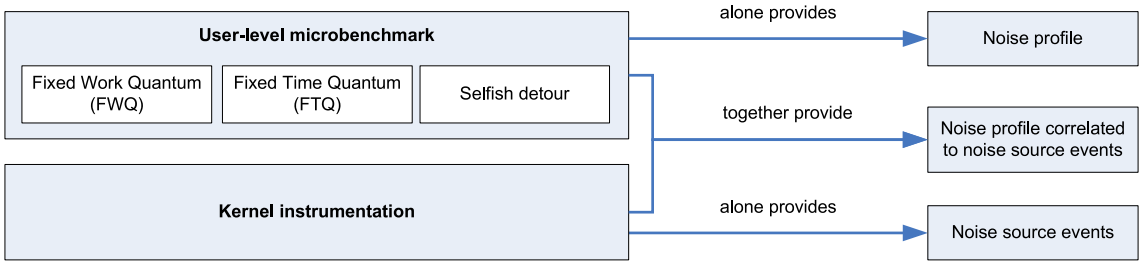


Figure 3.1: High-level categorisation of OS noise measurement approaches and their outputs. The two main approaches are *user-level micro-benchmarks* and *kernel instrumentation*. The two approaches can be utilised individually or in combination.

3.2.2 Micro-benchmark accuracy

Despite the fact that most OS noise measurement techniques utilise micro-benchmarks, very little existing work explicitly explores the effects of OS noise on micro-benchmark accuracy (Staelin, 2005; Staelin & McVoy, 1998).

Staelin and McVoy (1998) describe *mhz*, a micro-benchmark program designed to determine the processor clock speed in a system-independent manner. A series of techniques are utilised to ascertain the resolution and accuracy of the system clock measurement. *mhz* identifies timing errors by measuring the execution time of small variations on a processing task and determining if the variations in execution time match the corresponding variations in processing (Staelin & McVoy, 1998). A series of increasing processing granularities is tested, and the smallest granularity whose timings display less than 1% deviation from the expected variations is selected. The authors note that the system may be susceptible to noise and state that the median of all observations is utilised in place of the mean, in order to be more robust in the presence of outliers (Staelin & McVoy, 1998).

Staelin (2005) describes *lmbench*, an extensible micro-benchmark suite that includes *mhz* as one of its components. The same adaptive technique for determining clock resolution is used as described by Staelin and McVoy (1998). The author notes that timing results are often heavily skewed, containing extreme outliers that result in long tails in the distribution of the data. The majority of values are reported to cluster around the median (Staelin, 2005). Enhancements to *lmbench* to more thoroughly deal with the effects of noise are left as future work.

Existing micro-benchmarking tools do not sufficiently take into account the effects of OS noise, taking only simple measures such as reporting the median instead of the mean. A more comprehensive approach is

required that correctly measures and adaptively mitigates the effects of OS noise to maximise the accuracy of measuring fine-grained computational tasks. I develop such a technique through the case study that follows.

3.3 Case study: Apple iOS device OS noise profiles

This study develops a micro-benchmarking approach that adaptively mitigates the effects of OS noise, whilst maintaining the highest level of noise-reduced accuracy possible. I focus on the specific context of mobile device platforms.

As little existing research examines OS noise in the mobile context, it is instructive to first gather noise profiles for Apple iPad devices and examine their characteristics. I select the popular Apple iOS platform for this study because the platform features several characteristics that make it interesting in the context of examining OS noise with regards to micro-benchmarking accuracy:

- iOS applications are written in programming languages that are compiled into native machine code. Applications for other popular mobile platforms such as Android are commonly written in languages that utilise an application-layer Virtual Machine (VM), such as Java. The significant complexity introduced by the Java application-layer VM makes micro-benchmarking Java code extremely difficult (J. Y. Gil, Lenz & Shimron, 2011).
- The XNU kernel at the core of the iOS operating system is a variety of "tickless" kernel (Levin, 2012), which has been proposed in the existing literature to reduce levels of OS noise due by utilising reduced numbers of OS clock ticks (Akkan et al., 2012; Tsafrir et al., 2005).
- The memory allocation system utilised by iOS handles allocations of memory blocks differently based on their block size. Block sizes are rounded up to either the nearest multiple of 16 or 4096, depending on the requested size, resulting in increased memory use when inefficient block sizes are requested (Apple Inc., 2013). Memory state influences memory access-related effects including cache misses, page faults, and TLB misses, all of which have been identified as sources of OS noise (De et al., 2007; Morari et al., 2011; Tsafrir et al., 2005).
- The iOS operating system runs on a fixed set of hardware configurations manufactured by Apple. This provides a homogeneous hardware and software configuration for use in my experiment. The use of a well-defined and readily available hardware model also improves the reproducibility of my experiment.

The methodology and results of my experimentation to examine levels of OS noise on iOS devices are described in the following sections.

3.4 Experimental methodology

To measure the level of OS noise present on real mobile devices, I utilise 20 Apple iPad Air devices. All devices are identical hardware models and run identical software, a clean installation of Apple iOS 7.1.1. The network connections are disabled on all devices, and the screen brightness and volume of each device is set to 100%. All devices' clocks are synchronised with a single time server using the Network Time Protocol (NTP) (Mills, 2010), and the experiment code is scheduled to execute simultaneously across all devices. No other user-facing applications are present in memory while the experiment code is executing.

I measure OS noise using a variation of the Fixed Work Quantum (FWQ) technique. As in the selfish detour technique, I minimise the amount of processing performed by recording timestamps in a tight loop (Beckman et al., 2006). However, I minimise processing further than previous approaches by manually unrolling the loop, removing the overhead of the loop header itself. Minimising the processing performed maximises the sampling rate and ensures accuracy is as high as possible (Hoefler et al., 2010).

Unrolling the timestamp recording loop requires a fixed number of loop iterations. This precludes use of the filtering mechanism utilised by the selfish detour technique, which repeats until a fixed number of samples that match the filtering criteria have been collected (Beckman et al., 2006). As a result, I store all timestamp deltas generated. Due to the limitations of running FWQ for extended periods of time (M. Sottile & Minnich, 2004), I select a fixed iteration count of 10000, as a compromise between sample size and reducing memory-related side effects. Accordingly, each trace represents the level of OS noise present on a device at a single point in time, as opposed to across an extended period.

To capture OS noise levels corresponding to different device states, I perform a number of variations on the experiment code. Each variation executes a different set of state-altering processing immediately prior to noise measurement. The variations are designed to determine the influence of memory access and duration of processing. Six variations are performed:

- *NoPrior*: used as a control group, this variation performs no processing prior to noise measurement.
- *MemorySmall*: this variation allocates a series of small blocks of memory with odd-numbered block sizes. Prime numbers greater than 2 are chosen for use as block sizes because they are guaranteed to never be multiples of 16 or 4096. These allocations represent use of inefficient allocation block sizes (Apple Inc., 2013).
- *MemoryBig*: this variation allocates a series of large blocks of memory with power-of-two block sizes. Powers of two greater than 512 are chosen for use as block sizes because they are guaranteed to be multiples of either 16 or 4096. These allocations represent use of efficient allocation block sizes (Apple Inc., 2013).
- *CPUSmall*: this variation generates prime numbers in the range 0 to $100\text{x}10^8$ using the open-source primesieve code library (Walisch, 2014). The range is selected to execute for approximately 60 seconds based on initial testing using the devices. This duration was selected arbitrarily to represent a small change in time.

- *CPUMedium*: this variation generates prime numbers in the range 0 to 1000x10^8 using primesieve. The range is selected to execute for approximately 10 minutes based on initial testing using the devices. This duration was selected arbitrarily to represent a moderate change in time.
- *CPULarge*: this variation generates prime numbers in the range 0 to 4000x10^8 using primesieve. The range is selected to execute for approximately 40 minutes based on initial testing using the devices. This duration was selected arbitrarily to represent a large change in time, within the time constraints faced when performing the experiment.

Note that for the three CPU-bound variations, a loop and a fixed-size memory buffer is employed. This ensures that the differences in processing time do not result in differences in memory state, as memory is pre-allocated prior to any processing being performed.

Each variation is run twice: once when the devices are fully charged and connected to a power source, and once when the devices are disconnected from a power source and their batteries are discharging. This entire process is repeated for validation, resulting in four total runs consisting of two matched pairs of charged and discharging runs. The device battery level is fully charged when the power-disconnected runs commence, such that the processing time and complexity of each variation equates to the corresponding drain in battery level.

3.5 Experimental results

3.5.1 Outlier removal

The raw timestamp deltas collected for all variations of the experiment demonstrate similar characteristics to those noted by Staelin (2005). Data is clustered heavily around the median, and contains extreme outliers. These characteristics are demonstrated by large mean values for both skewness and kurtosis, shown in Table 3.3. This necessitates the removal of outliers from the data before it can be used to characterise levels of OS noise.

Table 3.3
Mean skewness and kurtosis of the collected data, before and after the removal of outliers.

	Raw data	**Outliers removed**
Mean skewness	26.12	0.36
Mean kurtosis	1076.55	5.54

I use as the basis for my outlier removal the interquartile range rule, the formula that underlies the box-plot, presented by Tukey (1977). This rule classifies all data that falls within the interquartile range, defined as the 75th percentile minus the 25th percentile, as inliers. This range of guaranteed inliers is then used to calculate cutoff points on either side for use in identifying outliers. Values are identified as potential outliers if they fall outside the cut-off points defined as follows:

$$\text{Lower cutoff} = Q1 - 1.5(Q3 - Q1)$$

$$\text{Upper cutoff} = Q3 + 1.5(Q3 - Q1)$$

The interquartile range rule is designed for identifying potential outliers on both the left and right sides of normally distributed data (Hubert & Vandervieren, 2008). The collected micro-benchmarking data has a hard lower bound: the execution time in the case that the level of OS noise is zero. As such, I treat all values to the left of the median as inliers. This approach is utilised in other fields when data has a hard lower bound, such as salary data in economics (Jenkins & Van Kerm, 2006). Given that I eliminate the lower cut-off, the upper cut-off can be modified to utilise the increased range of guaranteed inliers, as such:

$$\text{Upper cutoff} = Q3 + 1.5(Q3 - Q0)$$

Where $Q0$ is the 0th percentile, i.e. the minimum value in the dataset. This allows me to exclusively target outliers to the right of the data.

In addition to being characterised by a hard lower bound, the collected data is also skewed. The skewness of the data varies greatly. I use the *medcouple* robust measure of skewness (Brys, Hubert & Struyf, 2004) to determine the skew of the collected data without the influence of outliers. Possible values for the medcouple range from -1 to 1. Negative values represent left skewness, whilst positive values represent right skewness. A value of zero indicates a symmetrical distribution (Brys et al., 2004). I observe medcouple values for the collected data ranging between -1 and 0.98, spanning almost the entire range of skewness from one extreme to the other.

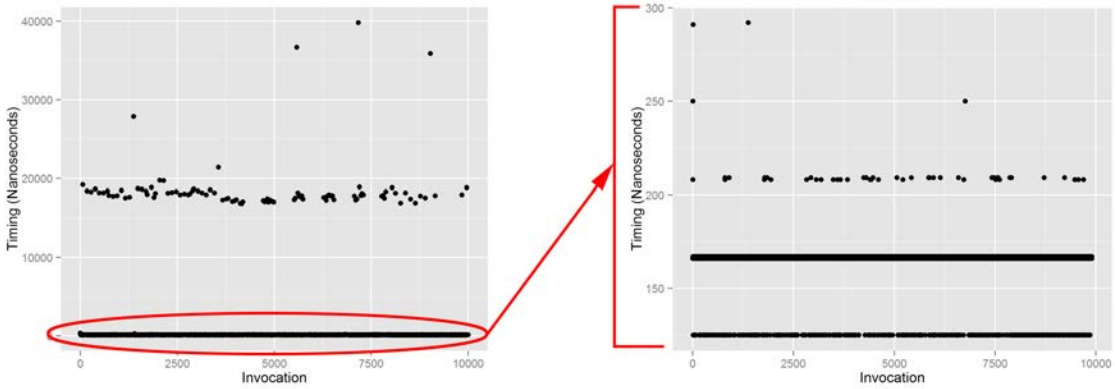
As noted by Hubert and Vandervieren (2008), the interquartile range rule is unsuitable for skewed data. A modification to the original formula is proposed for use with skewed data, known as the adjusted box-plot. However, the models utilised by the adjusted box-plot are unsuitable for extremely skewed data, which the authors specify as having a medcouple value less than -0.6 or greater than 0.6 (Hubert & Vandervieren, 2008). Applying the adjusted box-plot to the collected micro-benchmarking data produces unsuitable cut-off values in cases where skewness falls at the extreme ends of the scale.

After testing current outlier detection techniques, I find that no automated method provides consistently acceptable results across my entire collected dataset. I therefore develop a semi-automated process to perform outlier detection for my collected data. I utilise a systematic trial-and-error process, whereby outlier cut-off points are automatically generated for my entire dataset over varying percentiles and distance ratios. The results of the automated process are then verified using visual inspection over a set of exemplar datasets that represent varying values for skewness. The entire process is iterative, increasing the set of exemplar datasets as the list of candidate permutations is reduced. Finally, a single permutation remains, which represents the

most consistently acceptable outlier cut-off value. The final formula resulting from my analysis (and thus used to remove outliers from the collected micro-benchmarking data) is:

$$\text{Upper cutoff} = 95th\,percentile + 3.0(95th\,percentile - Q0)$$

Outlier removal using this formula produces the results displayed in the second column of Table 3.3. Although this formula produces acceptable results for the collected data, it is heavily parameterised based on the unique characteristics of this data and thus unlikely to be suitable for other datasets. Scatter-plots of an example dataset before (a) and after (b) outlier removal are depicted in Figure 3.2.



(a) (b)

Figure 3.2: Example scatter-plots of the collected micro-benchmarking data before (a) and after (b) outlier removal. Note the difference in scale: 0 - 40,000 for the raw data, 0 - 300 for the data after outlier removal.

3.5.2 Data analysis

Once outliers have been removed, I treat the range of the remaining data as a representative value of the level of OS noise. I select the range as a simple, easily computed measure of the presence and magnitude of observed performance variations. The use of a single value for OS noise level facilitates straightforward comparisons between results.

To identify the influence of device state on levels of OS noise, I first categorise the collected data based on device battery state. The data for each of the two battery states is then further categorised based on my six processing variations, as well as by device. I use Tukey's method to identify statistically significant differences in mean OS noise level between groups, at a significance level of 0.05.

Device For both battery states, comparison between the 20 devices identified no significant differences. This result is consistent across both matched pairs of charged and discharging runs. This confirms that the individual iPad devices were not influencing levels of OS noise differently to one another, as is expected of devices with identical hardware and software configurations.

Battery state When comparing battery state, no significant difference is found between devices connected to a power source and devices whose batteries are discharging. This result is consistent across both matched pairs of charged and discharging runs. Although the influence of battery charge level is not examined, the result support the conclusion that the presence or absence alone of a connected power source does not influence OS noise levels.

Memory allocation The results for the matched pairs of charged and discharging runs are inconsistent with regard to the influence of memory allocation. A significant difference is identified between the *MemoryBig* and *MemorySmall* allocations for the charged run of the first pair. No significant difference is identified between *MemoryBig* and *MemorySmall* in the discharging run of the first pair or either runs of the second pair. However, in all runs, a significant difference is identified between the two memory-allocating variations and the control group *NoPrior*. Although no conclusions can be drawn regarding the influence of allocation block sizes, the results suggest that performing allocations, regardless of block size, influences the level of OS noise.

Processing For both battery states, no significant difference is identified between the *CPUSmall*, *CPUMedium*, and *CPULarge* variations. However, for all runs, a significant difference is identified between the three processing variations and the control group *NoPrior*. This demonstrates that performing processing influences OS noise levels when compared to performing no processing, but that the length of processing has no influence. This suggests that the three processing times selected have the same impact on device state.

In the context of micro-benchmarking accuracy, the results discussed above highlight one key finding. The significant differences between the control group *NoPrior* and the other five variations demonstrate that OS noise levels change over time as the device state changes. This informs the OS noise level measurement strategy of my adaptive micro-benchmarking approach.

As OS noise levels appear to change over time, it is insufficient to perform a single measurement for use over the lifetime of a benchmark. Repeated measurements are required in order to ensure my adaptive mechanism maintains accurate information on current noise levels. The details of my adaptive micro-benchmarking approach are described in the section that follows.

3.6 Adaptive noise mitigation approach

My proposed adaptive noise mitigation approach for micro-benchmarking is in the spirit of the adaptive clock resolution approach of *mhz* (Staelin, 2005; Staelin & McVoy, 1998). The core principle of the approach is to determine the relevant characteristics of the system the micro-benchmark is executing on, and adaptively adjust the granularity of time measurements. Accuracy is maximised by identifying the finest level of granularity that masks measurement errors caused by the underlying system (Staelin & McVoy, 1998).

The system clock measurement mechanism I use under iOS has a resolution of nanoseconds (Apple Inc., 2005). Adjusting the granularity of time measurements is as simple as dividing all timestamp deltas by a scale

factor F, such that the unit of measurement utilised by my micro-benchmark becomes F nanoseconds. The fractional result of the division is truncated, masking any noise with a granularity less than F nanoseconds. Note that the raw time measurements used to derive the deltas are not transformed, only the timestamp deltas themselves. This prevents loss of accuracy due to unintended propagation of truncation. As the transformation of the data is driven purely by the scale factor, the value of F must be carefully selected.

To completely mask the level of OS noise present on the system, F should be equal to the range of the recorded noise. All deltas less than F are transformed to zero, eliminating all variations whose granularity falls within the range observed during noise measurement. However, this strategy suffers from a serious limitation. In all of the collected micro-benchmarking data, the vast majority of values cluster heavily around the median. In cases where the range of the observed noise is greater than the value of the median, the majority of the measurements for extremely fine-grained tasks will be transformed to zero. This violates the goal of maximising the accuracy of measurements.

An alternative strategy is to use the value of the median as the value of F. This ensures that maximum accuracy is maintained, as the majority of timestamp deltas for extremely fine-grained tasks will still report a non-zero value. In cases where the value of the median is smaller than the range of the noise, this logically works well. However, in cases where the value of the median is greater than the range of the noise, the granularity of measurements is coarser than it could be, given that the use of the range as the value of F would result in a finer granularity with no loss of accuracy.

In line with the goal of adaptivity, I propose a dynamic strategy that adapts to the characteristics of the data in question. The value of F should be either the range of the noise *or* the value of the median, whichever is smaller. This strategy ensures that both goals are met. The system always uses the finest granularity that satisfies the constraint of maximising accuracy.

To validate that the dynamic strategy for selecting the value of F is indeed an improvement, I compare the strategies by applying them to the data collected from my experiment. The results are shown in Table 3.4.

Table 3.4
Comparison of strategies for selecting the value of the scale factor F.

	Scale factor selection strategy		
	F =**Range**	F =**Median**	F =**min(Range, Median)**
Rate of median truncation to 0 (% of datasets)	89	0	0
Mean standardised difference between N and Range	0	0.57	0.51

The first row lists the percentage of datasets whose medians were truncated to zero by performing transformation using the scale factor. Zero percent is the ideal result for this metric. The strategy of unconditionally using the noise range as the scale factor value demonstrates its inherent flaw with a very large percentage of datasets whose medians are truncated to zero. The other two strategies do not truncate the median to zero for any of the datasets.

The second row lists the mean difference between the selected scale factor value and the minimum possible

granularity, which is the noise range. The absolute difference between the scale factor value and the range is standardised with respect to the range. A lower value is better for this metric, with zero representing the ideal result. The strategy of unconditionally using the noise range as the scale factor shows a difference of zero, as the range and the scale factor are identical. The strategy of unconditionally using the median as the scale factor shows the highest difference, as the median is selected even in instances where the noise range could be used without loss of accuracy. The adaptive strategy shows a lower difference than the strategy of unconditionally using the median, as the range is selected in instances where it can be used without loss of accuracy.

The results demonstrate for the collected micro-benchmarking data that the dynamic strategy for selecting the value of the scale factor F provides an improved compromise between minimising granularity and preventing the truncation of the median to zero. Transforming timestamp deltas using the scale factor value selected by this strategy allows my proposed approach to adaptively mitigate the effects of OS noise on micro-benchmarks, whilst maintaining maximum accuracy.

3.7 Discussion and future work

Performance benchmarking is commonly used to evaluate the merits of technological products, and can influence consumer perceptions. In the mobile context, benchmarking is prominent enough to have received attention from device vendors (AnandTech, Inc., 2013). Micro-benchmarking is a subset of benchmarking that examines individual components of an application instead of the whole. Analysis of individual components is a useful tool for improving application responsiveness, which greatly influences user experiences of interactive applications. Improvement to application analysis tools, such as micro-benchmarking, provides software developers with flexibility and further facilitates the delivery of mature and robust mobile products and interactive consumer applications.

Micro-benchmarking receives little focus in the mobile context compared to broader-grained benchmarking. This is likely due to the limitations of existing micro-benchmarking techniques, including susceptibility to the effects of OS noise. Current micro-benchmarking techniques do not provide a comprehensive method of mitigating the effects of OS noise on micro-benchmark results. Existing micro-benchmarks frequently repeat operations a large number of times, measuring the overall timestamp delta. This coarse-grained measurement is then divided by the iteration count to determine the mean iteration time (Staelin, 2005). Even when this process is repeated and the median of the means is reported (Staelin & McVoy, 1998), this approach precludes the detection and removal of all outliers because individual iteration times are not recorded. The results of this naive approach are not fully robust to outliers, and are inflated in the presence of extreme outliers.

My proposed adaptive noise mitigation technique adapts to the levels of OS noise present to mitigate the effects on micro-benchmark results. By measuring individual iteration times and removing outliers, an accurate profile of OS noise levels present is generated. The dynamically selected transformation strategies utilise the finest level of measurement granularity available that mitigates the effects of OS noise levels.

Mitigating the effects of OS noise using the proposed technique allows micro-benchmarks to achieve higher levels of accuracy than previous techniques and addresses one of the most significant limitations of current micro-benchmarks. This enhances the utility of micro-benchmarking as a fine-grained performance measurement tool, presenting opportunities for micro-benchmarks to be utilised in a broader range of contexts than before. The utility of micro-benchmarking is expected to increase even further through the continued expansion of my research.

This study is a preliminary exploration of OS noise on Apple iPad Air devices. Future expanded studies are anticipated to include a broader number of mobile devices and platforms. In addition to a greater set of hardware and software configurations, an expanded set of device state information can also be collected. A greater variety of processing durations, with differences in the order of hours, captures data representative of a full battery discharge cycle. Utilising an expanded set of state information is likely to provide greater insight into the relationship between OS noise levels and device state changes.

Expanded studies present potential for further validation and development of my adaptive noise mitigation technique. Validation over a broader range of mobile device configurations facilitates evaluation of the applicability of my adaptive noise mitigation technique to a more generalised context. The iterative, semi-automated process utilised for outlier removal also presents potential for further development toward a fully automated process. I describe such an automated process in Chapter 4.

3.8 Conclusion

Micro-benchmarking is a useful tool in the fine-grained optimisation of mobile platforms and applications. Micro-benchmarking provides developers with additional flexibility in delivering robust mobile products and consumer applications. Existing micro-benchmarking implementations are improved through mitigation of the effects of OS noise.

I conduct a study into the levels of OS noise on real-world mobile devices. The collected data demonstrates large variations in the distribution of noise data, and includes extreme outliers. I develop a semi-automated approach for removing outliers from the data prior to performing analysis. Analysis results demonstrate changes in OS noise levels due to changes in device state. I then propose an adaptive noise mitigation technique informed by the findings from my study.

My adaptive noise mitigation technique dynamically selects a strategy for transforming the granularity of measurements. OS noise level data is utilised to select the smallest granularity that matches the transformation correctness constraints. This approach adaptively mitigates the effects of OS noise on micro-benchmark results, whilst maintaining the maximum possible noise-reduced accuracy.

Chapter 4

Automated outlier removal

This chapter builds on the adaptive noise mitigation technique proposed in Chapter 3 to completely address research objective 4 by fully automating the outlier removal process. In this chapter, I collect OS noise data from various Apple iOS device models and examine common characteristics of the collected data. Based on the nested clustering structure evident in the collected data, I propose an automated outlier technique that utilises hierarchical clustering for removing outliers from micro-benchmarking datasets. A heuristic is proposed for automatically determining the height at which to cut the dendrogram produced by hierarchical clustering, so that outliers can be removed based on the resulting clusters. I then propose a simplified version of this heuristic that utilises pre-computed data to reduce computational requirements and make outlier removal feasible on resource-constrained mobile devices.

Although the outlier removal heuristic for cutting the dendrogram produced by hierarchical clustering is computationally inexpensive, the hierarchical clustering process itself dominates the time complexity for performing outlier removal. This limitation is addressed in Chapter 5, which proposes an efficient parallel algorithm for performing hierarchical clustering.

The content from this chapter has been published as:

- *Rehn, A., Holdsworth, J., & Lee, I. (2015). Automated outlier removal for mobile micro-benchmarking datasets. In* 10th international conference on intelligent systems and knowledge engineering (ISKE) *(p. 578-585).*

4.1 Introduction

Micro-benchmarking is the performance analysis of individual components or small subsets of an application (Staelin, 2005). Just as application-wide benchmarking is useful in evaluating the real-world performance

of entire applications, micro-benchmarking can aid developers in measuring the performance of individual application components. Such components could include the responsiveness of user interface interactions, which have been shown to play a significant role in users' perception of mobile applications (Tolia et al., 2006). As such, micro-benchmarking represents a useful tool in the development of mobile platforms and applications.

Nevertheless, fine-grained measurement on small time-scales presents additional complexities that must be addressed in order to produce meaningful results. The inherent nature of modern hardware and operating systems introduces noise into micro-benchmarking measurements (S. Wang et al., 2002). This noise can include outliers of varying levels of magnitude. Existing micro-benchmarking approaches frequently utilise naive approaches to counteract the influence of outliers (Staelin, 2005; Staelin & McVoy, 1998). Even though there exist many studies in general outlier detection (Breunig, Kriegel, Ng & Sander, 2000), I am aware of no existing work that explores a fully unsupervised method of removing outliers from micro-benchmarking data in a mobile context.

In this chapter I examine micro-benchmarking data collected from numerous mobile devices and identify common characteristics. I propose an automated algorithm for identifying outliers in micro-benchmarking datasets. The heuristic algorithm is based on the observed characteristics of the collected mobile micro-benchmarking datasets. I then develop a simplified heuristic suitable for use on mobile devices.

The contributions of this study are as follows:

- I analyse the characteristics of micro-benchmarking data collected from a variety of mobile devices. I am not aware of any previous study that provides a detailed discussion of micro-benchmarking data characteristics across a range of mobile device hardware;
- The proposed automated outlier detection algorithm is highly amenable to simplification for use on resource-constrained mobile platforms. Evaluation using a number of existing metrics demonstrates the effectiveness of my algorithm;
- The simplified variation of the proposed algorithm runs in $O(n \log n)$ time, making it highly scalable and suitable for use on mobile devices with limited resources. This allows automated outlier removal to be performed on-device, providing meaningful results without the need to transfer collected data to another device for post-processing.

The rest of the chapter is organised as follows. First, I provide a background on the cause of noise in micro-benchmarking data and discuss how existing approaches deal with this noise. I motivate my research by describing the unique considerations that must be taken into account on mobile platforms, and examine the characteristics of collected mobile micro-benchmarking datasets. I then describe my proposed outlier removal algorithm and evaluate its effectiveness using existing metrics. Finally, I develop a simplified variation of the proposed outlier removal algorithm, and evaluate both its effectiveness and computational efficiency in comparison to the full algorithm.

4.2 Related work

Noise in micro-benchmarking data arises from the complexities of measuring the wall-clock performance time of extremely fine-grained tasks. Modern consumer operating systems are designed heavily for multitasking. Processors must constantly switch between running applications, system services, and servicing hardware communication. This results in small perturbations to application performance. The performance variation caused by system activity is referred to as OS noise (Akkan et al., 2012). OS noise arises from numerous sources, which are discussed in Section 3.2.1.1. The magnitude of OS noise is typically on the order of sub-second time-scales (Morari et al., 2011; Tsafrir, 2007), and hence of importance primarily when performing extremely fine-grained performance measurements.

OS noise only affects the wall-clock performance time of applications, and has no effect on other measurements such as CPU time. CPU time measures the number of CPU cycles spent executing both user-mode and kernel-mode instructions for a process. This measurement excludes time spent executing other processes, thus discarding the influence of OS noise. CPU time is commonly used by profiling tools, whose purpose is to analyse the relative complexity of an application's components with respect to one other and identify performance bottlenecks (Graham, Kessler & Mckusick, 1982). In contrast, the purpose of benchmarking and micro-benchmarking is to measure the real-world performance of an application on a specific hardware configuration. Accordingly, micro-benchmarking tools utilise wall-clock time, resulting in their susceptibility to the effects of OS noise.

Most existing micro-benchmarking tools utilise only extremely simple approaches for addressing the effects of OS noise. The tools *mhz* (Staelin & McVoy, 1998) and *lmbench* (Staelin, 2005) both implement robustness against outliers by reporting the median of measured times instead of the mean. In the mobile context, *AndroBench* (Kim & Kim, 2012) utilises the three sigma rule to remove outliers. The use of this heuristic implies the assumption of a normal distribution, which is rarely the case for benchmark timing data (Staelin, 2005). The most sophisticated outlier removal I observe is that of the micro-benchmarking tool by Jamal and Waheed (Jamal & Waheed, 2008), which utilises k-nearest neighbour clustering to remove outliers. However, this tool is designed purely for traditional desktop platforms, not for a mobile context.

There is a clear lack of micro-benchmarking tools for mobile platforms that utilise a comprehensive approach to outlier removal. The development of an outlier removal approach for mobile micro-benchmarking must take into account the constraints of mobile platforms. Limited memory and processing power dictate that computational complexity and memory usage be kept to a minimum. This precludes the use of complex algorithms. I propose that a heuristic based on the characteristics of the data being processed is best suited to satisfying the constraints of micro-benchmarking on mobile devices. In the following section, I analyse micro-benchmarking datasets that I collect from real mobile devices, and analyse the common characteristics of the data.

4.3 Data characteristics

To analyse the characteristics of mobile micro-benchmarking data, I collect micro-benchmarking datasets from a series of real mobile devices. My collection harness measures the levels of OS noise present on a device by micro-benchmarking the performance of querying the system time. For consistency, I collect data from devices all running the Apple iOS operating system. These devices include the fourth-generation iPod Touch, the iPhone 5, and both the third-generation iPad and fourth-generation iPad Air.

Each dataset contains 5000 samples. After initial testing with various sample sizes, I select this sample size as a trade-off that allows me to collect enough data to perform meaningful analysis, whilst not exceeding the limited memory capacities of the older device models. The collected data possesses a number of characteristics that instruct the choice of an outlier removal approach. These characteristics include the data distribution and clustering structure, and the types of outliers present.

The distribution of the data varies greatly between datasets. As in Chapter 3, I measure the skewness of each dataset using the medcouple, a robust measure of skewness (Brys et al., 2004). Recall that medcouple values range from -1 to 1. Negative values indicate a distribution that is skewed to the left, whilst positive values indicate a distribution that is skewed to the right. A medcouple value of zero indicates a symmetric distribution (Brys et al., 2004). The medcouple values for the collected datasets span the full range possible, demonstrating varying levels of skewness to both the left and right. This variation precludes the use of distribution-based outlier detection approaches.

Visual inspection of the collected data reveals that all of the datasets contain multiple clusters of varying density. I observe the majority of data is typically contained in a small number of extremely dense clusters, which are often nested within a surrounding cluster of a lower density. In the majority of instances, the innermost cluster contains the median, which is consistent with existing observations of micro-benchmarking data (Staelin, 2005). The density of the clusters containing the inliers is largely consistent among mobile devices of the same hardware model, but varies across different hardware models. Figure 4.1 depicts example datasets for several different mobile device models. The presence of clusters nested within other clusters necessitates the use of a hierarchical clustering algorithm, which represents these nested relationships.

I observe two distinct types of outliers in the collected data. The first type of outliers are data points that clearly do not belong to any cluster. I refer to these as “standalone” outliers. The second type are data points that form small, low-density clusters. These clusters are extremely small with respect to the sample size, and have a low density when compared to the clusters that contain the majority of the data. I refer to these as “clustered” outliers.

The distinction between the two types of outliers is based on visual inspection. To confirm this distinction as being meaningful, I validate it using the existing metric of Local Outlier Factor (LOF) (Breunig et al., 2000). LOF measures the extent to which each data point is isolated from its local neighbourhood and assigns it a value to represent the degree to which it is an outlier (Breunig et al., 2000). The local nature of LOF makes it ideally suited to identifying noise points that are distant from the nearest cluster. LOF has one parameter, *MinPts*, which determines the number of surrounding points that constitute a point’s local neighbourhood.

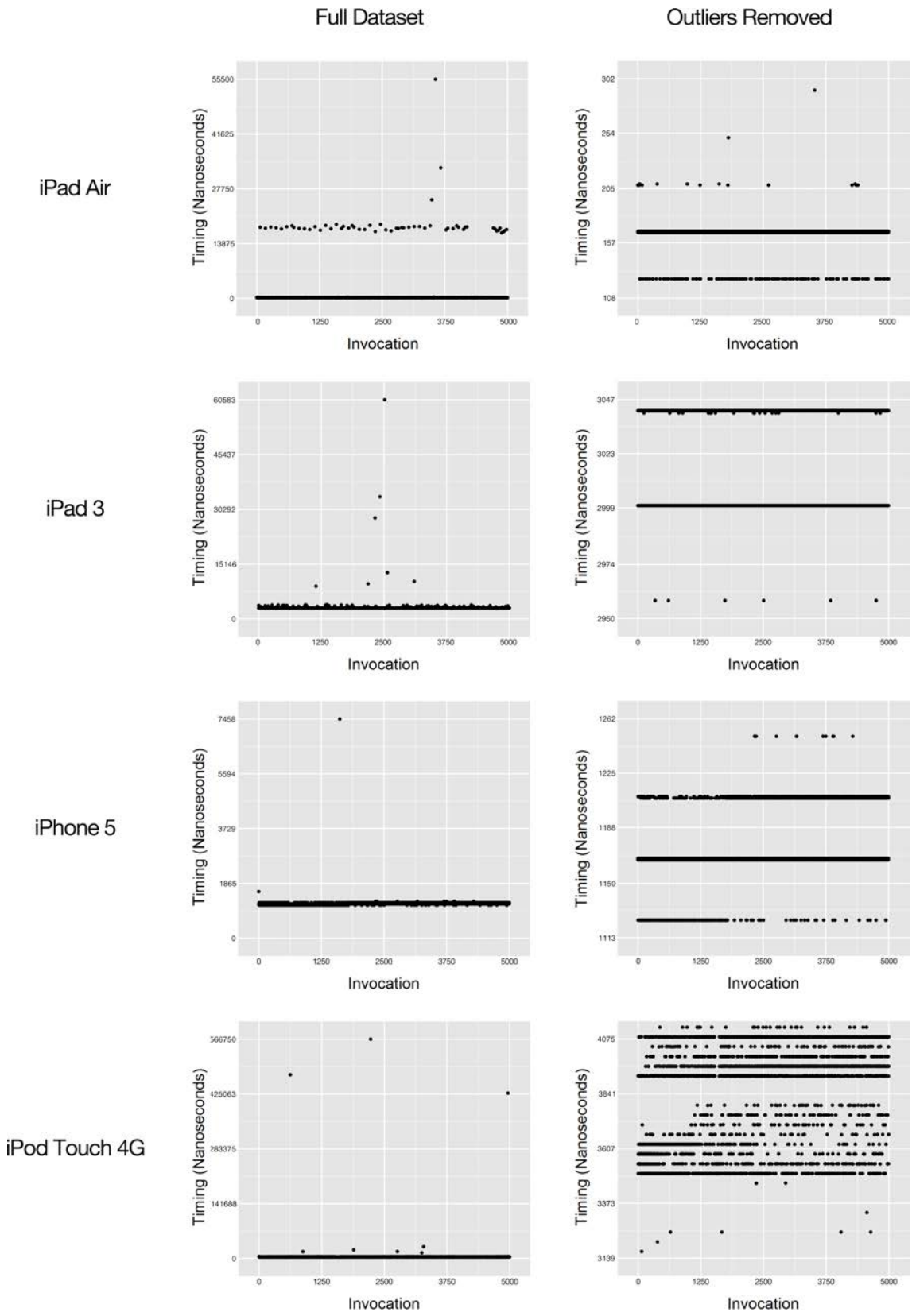


Figure 4.1: Example datasets for each of the mobile device hardware models. The top row depicts each dataset in full, whilst the bottom row depicts each dataset once outliers have been removed.

When computing the LOF values for the points in the collected datasets, I use a *MinPts* value of 10, which is the lowest value recommended (Breunig et al., 2000). I find that the points identified as standalone outliers

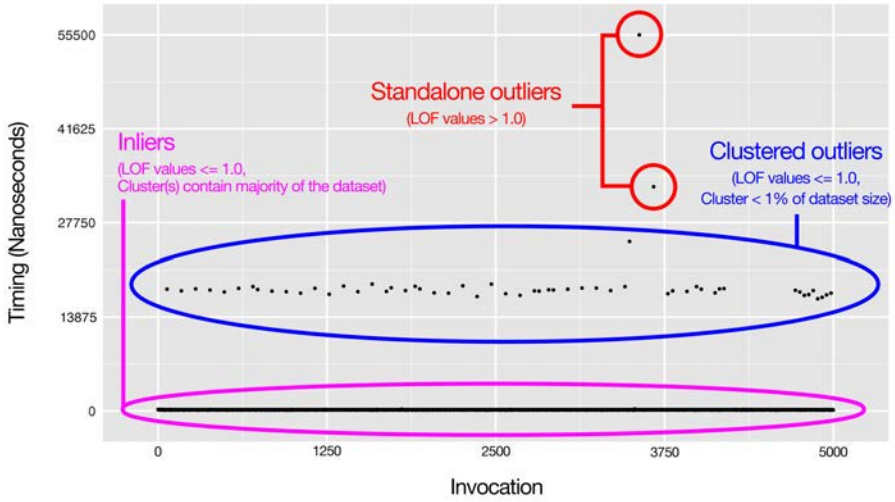


Figure 4.2: Example dataset highlighting both inliers and the two types of encountered outliers.

are consistently assigned LOF values greater than 1.0, whilst clustered outliers are consistently assigned LOF values ≤ 1.0. Further, I observe that in the majority of cases there exists at least one border point in each outlier cluster that is identified as a standalone outlier, based on its LOF value.

Formally, I define inliers and the two types of outliers as such:

Definition 1 *For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, a **standalone outlier** is a point $p_i \in P$ not belonging to any cluster, having an LOF value > 1.0.*

Definition 2 *For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, a **clustered outlier** is a point $p_i \in P$ belonging to a cluster with a size < 1% of the dataset size ($|P|$), having an LOF value ≤ 1.0.*

Definition 3 *For a given $P = \{p_1, p_2, \ldots, p_n\}$ set of points, an **inlier** is a point $p_i \in P$ belonging to a cluster with a size > 1% of the dataset size ($|P|$), having an LOF value ≤ 1.0.*

Examples of inliers and the two types of outliers are illustrated in Figure 4.2.

In a small number of instances, I observe data points that resemble standalone outliers in the left tail of the dataset. It is important to note that these points should not be considered outliers. For any given task, the minimum possible time in which it can be performed is determined by the processor speed. OS noise can only increase the amount of time taken to perform processing above this minimum. Accordingly, although there is no theoretical upper bound to the performance timing of a task, there is always a hard lower bound (Staelin, 2005). The only exception to this rule is when the system or device is not in a steady-state (J. Y. Gil et al., 2011). However, my data collection methodology ensures all of my datasets are collected during steady-state operation.

4.4 Outlier removal algorithm

Due to the presence of nested cluster relationships within the collected data, a hierarchical clustering algorithm is necessary to capture the clustering structure (Jain, Murty & Flynn, 1999). I select agglomerative hierarchical clustering using the complete linkage metric. Complete linkage is selected because it is less sensitive to the presence of outliers than other metrics such as single-linkage (Jain & Dubes, 1988), and due to the availability of an efficient implementation (Murtagh, 1983). I utilise Minkowski distance as the dissimilarity metric, which is equivalent to Manhattan distance in this context, as my data is univariate (Kaufman & Rousseeuw, 1990).

Agglomerative hierarchical clustering forms clusters from data points by iteratively increasing the neighbourhood distance at which objects are considered to be part of the same cluster (Murtagh, 1983). This process halts once a neighbourhood distance has been reached that results in the entire dataset being placed in a single cluster. The output of hierarchical clustering is a data structure known as a dendrogram. Dendrograms are tree structures, wherein the leaf nodes represent the individual data points and each interior node represents a cluster containing all of the leaf nodes that are its descendants (Jain et al., 1999). Each interior node is marked with a "height", which is the neighbourhood distance at which the cluster that the node represents was formed. The root node of the tree represents a single cluster containing the entire dataset (Kaufman & Rousseeuw, 1990).

To extract clusters from a dendrogram, the tree is typically "cut" at a particular height. Cutting a dendrogram at a given height produces the clustering corresponding to the neighbourhood distance represented by the height (Jain et al., 1999). Since a dendrogram represents the entire hierarchical clustering structure of a dataset, the choice of cut height significantly impacts the number of extracted clusters. Cuts closer to the root of the tree will result in fewer clusters, whilst cuts closer to the leaves will result in a greater number of clusters. The choice of cut height is typically specific to the data being clustered, and is often selected manually (Kaufman & Rousseeuw, 1990). Automating the choice of cut height allows the clustering process to become fully unsupervised. To automate the choice of cut height for mobile micro-benchmarking datasets, I develop a heuristic that takes into account the observed characteristics of the data.

I aim to satisfy the following goals in developing my outlier detection heuristic:

- Outlier removal should be fully automated and require no manual intervention;
- Both standalone and clustered outliers should be removed; and
- Any complex analysis performed on the data should be able to be precomputed, in order to achieve simplified complexity when performing outlier removal on mobile devices.

The last goal is of particular importance, as it restricts the types of processing that can be performed. It precludes the use of complex algorithms whose output cannot be used to discover relationships between the underlying data and the optimal choice of cut height. A number of existing approaches for unsupervised extraction of clusters from dendrograms have been proposed (Almeida, Barbosa, Pais & Formosinho, 2007;

Algorithm 4.1 Full outlier removal heuristic.

Input: threshold - the minimum cluster size threshold
Input: dataset - the dataset that has been clustered
Input: D - the dendrogram, $D = (V, E)$
Output: the height at which to cut the dendrogram

```
1:  function HEURISTICFULL(threshold, dataset, D)
2:      Candidates ← ∅
3:      for v in V do
4:          if children[v] ≠ ∅ and height[node] ∉ Candidates then
5:              clusters = CUTDENDROGRAM(D, height[node])
6:              sumLOF ← 0
7:              numSamples ← 0
8:              for cluster in clusters do
9:                  if |cluster| ≥ threshold or min(cluster) ≤ median(dataset) then    ▷ If cluster members are inliers
10:                     sumLOF ← sumLOF + LOF(cluster)
11:                     numSamples ← numSamples + |cluster|
12:                 end if
13:             end for
14:             cleanedLOF ← sumLOF ÷ numSamples
15:             Candidates ← Candidates ∪ {height[v], cleanedLOF}
16:         end if
17:     end for
18:     minLOF ← min(l | {h,l} ∈ Candidates)
19:     Candidates′ ← {h,l} ∈ Candidates | l = minLOF
20:     find {h,l} ∈ Candidates′ | h = max(x | {x,y} ∈ Candidates′)
21:     return h
22: end function
```

Langfelder, Zhang & Horvath, 2008; Sander, Qin, Lu, Niu & Kovarsky, 2003), but these algorithms focus on characteristics of the dendrogram alone and not on relationships between the dendrogram and the underlying dataset. Ensuring that any discovered relationships can be simplified into a cheaply computed representation facilitates performing outlier removal on actual mobile devices. The ability to perform outlier removal on-device greatly increases its utility to mobile micro-benchmarking applications.

To ensure the removal of both standalone and clustered outliers, I utilise different methods for detecting each type. Standalone outliers are easily identified by computing the LOF values for a dataset. Based on my observations, those points with LOF values greater than 1.0 are standalone outliers. To remove clustered outliers, I utilise a threshold for the minimum size of a cluster. The points in any cluster that is smaller than the threshold value are marked as outliers. This threshold also removes standalone outliers when they are clustered alone or with neighbouring outlier clusters. To address the fact that outliers can only exist in the right-hand tail of the data, I only mark clusters as outliers when the minimum value in the cluster is greater than the median of the dataset. Based on analysis of the data, I set the minimum cluster size threshold at 1% of the size of the dataset.

To automate the selection of a dendrogram cut height, I exploit the relationship between standalone and clustered outliers. For almost all collected datasets, I observe that each outlier cluster contains at least one border point that is identified as a standalone outlier by its LOF value. Early in the agglomerative clustering process, these border points are clustered with their neighbouring outlier cluster. As the clustering process progresses, the outlier cluster itself is eventually clustered with other clusters, until the point at which it is clustered with inliers. By examining how many points with LOF values greater than 1.0 are marked as

Algorithm 4.2 The simplified version of the outlier removal heuristic.

Input: D - the dendrogram, $D = (V, E)$
Input: rankcurve - the curve with which to rank cut levels
Output: h - the height at which to cut the dendrogram

```
1:  function HEURISTICSIMPLIFIED(D)
2:      Heights ← ∅
3:      RankedHeights ← ∅
4:      for v in V do
5:          if children[v] ≠ ∅ and height[v] ∉ Heights then
6:              Heights ← Heights ∪ height[v]
7:          end if
8:      end for
9:      for height in Heights do
10:         cutLevel ← index[height] / |Heights|
11:         RankedHeights ← RankedHeights ∪ {height, RANKCURVE(cutLevel) }
12:     end for
13:     find {h, r} ∈ RankedHeights such that r = max(y | {x, y} ∈ RankedHeights) ▷ Find the height with max rank
14:     return h
15: end function
```

outliers, I can identify the stage at which outliers start to be clustered with inliers. The neighbourhood distance immediately preceding this stage represents the highest level of clustering that partitions outliers and inliers separately. I refer to this neighbourhood distance as the "ideal" cut height. It is this cut height that I wish to extract.

Definition 4 *The **ideal cut height** for a dataset is the greatest cut height which results in a clustering that partitions outliers and inliers separately.*

Once hierarchical clustering has been performed on a dataset, the algorithm to identify the ideal cut height is relatively straightforward. First, I extract the set of unique cut heights from the dendrogram. I refer to this set as the "cut candidates" for the dendrogram.

Definition 5 *The **cut candidates** for a dendrogram are the set of unique cut heights that exist in that dendrogram.*

I then sort the list of cut candidates. Sorting the list effectively assigns each cut candidate a ranking value equal to its position in the sorted list. The cut candidate with the best rank represents the ideal cut height. The pseudocode for the heuristic is listed in Algorithm 4.1.

4.5 Simplified heuristic for mobile devices

In the section that follows, I analyse the generated ranked lists and produce a simplified representation of the identified relationships for use on mobile devices. The use of a ranking-based approach was selected because the ranked list of cut candidates represents an output that can be further analysed to identify relationships within the collected data.

To determine if my outlier removal heuristic can be simplified for use on mobile devices, I analyse the ranked lists of cut candidates for the collected datasets. For each cut candidate, the number of outliers identified is recorded. I observe that ranges of neighbouring cut heights frequently identify the same number of outliers as one another. This suggests that in each dendrogram there exists a number of contiguous ranges of cuts whose clusterings can be treated as equivalent in regard to their outlier detection result. Most importantly, for most of the collected datasets I observe that there exists a range of cut heights that all identify the same number of outliers as the ideal cut height. The relationships between these ranges represent a potential simplification of my heuristic.

Before it is possible to identify any inter-dataset relationships, it is necessary to normalise the cut heights to produce values that facilitate comparison. Cut heights represent neighbourhood distances in the coordinate space of the distance metric used to compare points in the underlying dataset (Jain et al., 1999). The simplest approach is to normalise cut heights with respect to the maximum cut height present in their respective dendrogram. However, since this maximum is equal to the depth of the entire dataset, the resulting normalisation is affected by the scale of the underlying data. To alleviate this, I instead normalise cut heights based on their position in the list of unique cut heights extracted from their respective dendrogram. I refer to this normalised value as the "cut level". The use of cut levels instead of cut heights effectively rescales dendrogram space to eliminate any influence from the scale of the underlying data.

Definition 6 *A **cut level** is a cut height normalised with respect to its position in the ordered set of cut candidates.*

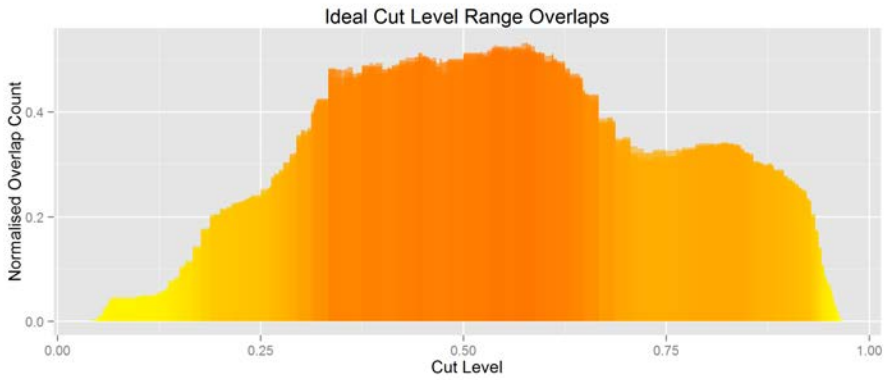


Figure 4.3: The unique intersections between the ideal ranges of the collected datasets. The y-value and colour saturation of each bar represents the normalised overlap count for that intersection.

For each dataset, I identify the upper and lower bounds of the range of cut levels that identify the same number of outliers as the ideal cut height. I refer to this as the "ideal range" for that dataset. I then compute the set of unique intersections of the ideal ranges for all of the collected datasets. For each intersection, I count the number of datasets whose ideal range is a superset of the intersection. I refer to this value as the "overlap count". The overlap count for an intersection represents how frequently the range of cuts represented by the intersection yields the same number of outliers as the ideal cut height. Figure 4.3 depicts the intersections' ranges plotted against their overlap counts, which have been normalised with respect to the number of collected datasets.

The visualisation of the overlap counts reveals a distribution that is largely symmetric around the centre, indicating that cut candidates whose cut level is between 0.3 and 0.6 most frequently result in the ideal number of outliers being detected. The presence of this relationship facilitates the simplification of my heuristic to a simple curve that ranks a list of cut candidates according to their cut level. Sorting can be replaced with simple evaluation of a curve value for each cut candidate. The pseudocode for this simplified, curve-based heuristic is listed in Algorithm 4.2.

In order to utilise the simplified heuristic, a curve must first be selected. I fit a sixth-degree polynomial to the silhouette of the overlap count plot, which results in a curve that follows the silhouette very closely. However, I expect that the overall maxima of the curve is more important than the rest of its shape, as the peak indicates the cut levels that will be ranked the highest. To validate this intuition, I use the standard bell curve of the normal distribution. This curve is simple to model, and its peak aligns closely with the silhouette of the overlap count plot. The two curves are shown in Figure 4.4.

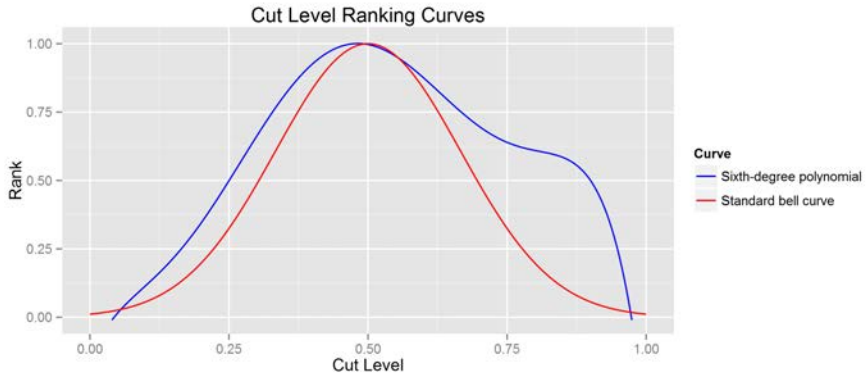


Figure 4.4: The two cut level ranking curves I evaluate for use with the simplified heuristic.

To determine which curve best represents the trends visualised by the overlap count plot, I compare the outlier detection results of the two curves with respect to the full heuristic. I rank the lists of cut candidates for all collected datasets using each curve in turn. The number of outliers detected by the highest ranked cut candidate is compared in each case to the number of outliers detected by the ideal cut candidate for that dataset. I then normalise the outlier detection error values with respect to the maximum possible outlier error, which is equal to the size of the dataset. The mean error values for both of the curves are listed in Table 4.1.

Table 4.1
Normalised mean outlier detection error values for the two curves.

Curve	Normalised mean outlier detection error
Sixth-degree polynomial	0.004356
Standard bell curve	0.004354

Interestingly, the mean error value for the standard bell curve is slightly lower than the mean error value for the polynomial curve. This result suggests that matching the exact silhouette of the overlap counts plot is less important than matching the overall distribution. This result also validates my expectation that the position and shape of the maxima of the curve are the most important features. The low mean outlier detection error

value indicates that the curve is indeed a fairly close approximation of the behaviour of the full heuristic.

4.5.1 Evaluation

4.5.1.1 Outlier removal effectiveness

In this section I validate the effectiveness of my outlier removal approach. For each collected dataset, I examine the original polluted dataset, the version of dataset after outlier removal using the full heuristic, and the version after outlier removal using the simplified heuristic. I treat each version of the dataset as a single cluster. This facilitates the use of existing measures of cluster compactness to quantify the effect of outlier removal. I compute values for the following measures:

- Depth (convex hull);
- Radius;
- Total Within-Group Distance (TWGD); and
- The absolute difference between the cluster's centroid and its medoid.

The values for these metrics are listed in Table 4.2. All four metrics demonstrate a large difference between the polluted versions of the collected datasets and the cleaned versions. Very little difference exists between the results for the full heuristic and the simplified heuristic. The results of these metrics confirm the effectiveness of my outlier removal approach, and also demonstrate that the simplified heuristic performs just as well as the full heuristic, when an appropriate cut ranking curve is utilised.

4.5.1.2 Algorithm efficiency

I determine the time complexity for both the full and simplified heuristic by performing theoretical complexity analysis on the algorithms' pseudocode. The time complexity for both algorithms is listed in Table 4.3. Note that the complexity listed excludes the construction of the dendrogram itself, which varies in complexity depending on the clustering algorithm used (Murtagh, 1983).

The full heuristic operates in $O(n^2)$ time in the worst case. This is because the dendrogram must be cut once for each unique cut height. Cutting the dendrogram has $O(n)$ complexity in all of the implementations I have tested. In the worst case scenario the number of unique cut heights in the dendrogram is equal to the size of the dataset. It is worth noting, however, that in all of my collected datasets, the number of unique cut heights was in fact quite small with respect to the size of the dataset. The worst case scenario can only occur when the set of pairwise distances between all points in a dataset contains at least as many unique distances as there are data points. This scenario is not typical of the observed characteristics of micro-benchmarking results, and is unlikely to be encountered in real usage.

The simplified heuristic operates in $O(n \log n)$ time in the worst case. This is because for each interior node, it is necessary to search the existing set of unique heights to determine whether or not to add the current

Table 4.2
Results for the four compactness metrics: TWGD, Depth, Radius, and the difference between the centroid and the medoid.

Metric	Polluted Dataset	
	Mean	Std. Dev
TWGD	2325035669.78	2860970117.61
Depth	41017.59	247786.01
Radius	1034.84	3719.54
Diff Centroid Medoid	58.55	113.78
	Cleaned Dataset (Full Heuristic)	
	Mean	Std. Dev
TWGD	598681922.73	526122255.17
Depth	381.37	1085.84
Radius	47.97	42.42
Diff Centroid Medoid	14.77	11.64
	Cleaned Dataset (Simplified Heuristic)	
	Mean	Std. Dev
TWGD	594372693.58	563944711.27
Depth	237.15	198.79
Radius	46.16	41.67
Diff Centroid Medoid	15.03	12.59

Table 4.3
Time complexity for the full and simplified outlier detection heuristics, excluding dendrogram construction.

Heuristic	Worst-case time complexity
Full	$O(n^2)$
Simplified	$O(n \log n)$

node's height to the set. I assume an implementation that can perform this search in $O(\log n)$ time. This search is also present in the full heuristic but is dominated by the complexity of cutting the dendrogram for each cut height.

In addition to performing theoretical complexity analysis, I measure the average runtime of the two heuristics for my collected datasets. The results of these measurements are depicted in Figure 4.5. The trends demonstrated by the average measured runtimes conform to the time complexities determined by my theoretical analysis. Note that, as is the case in the theoretical complexity analysis, the measured runtime values do not include dendrogram construction.

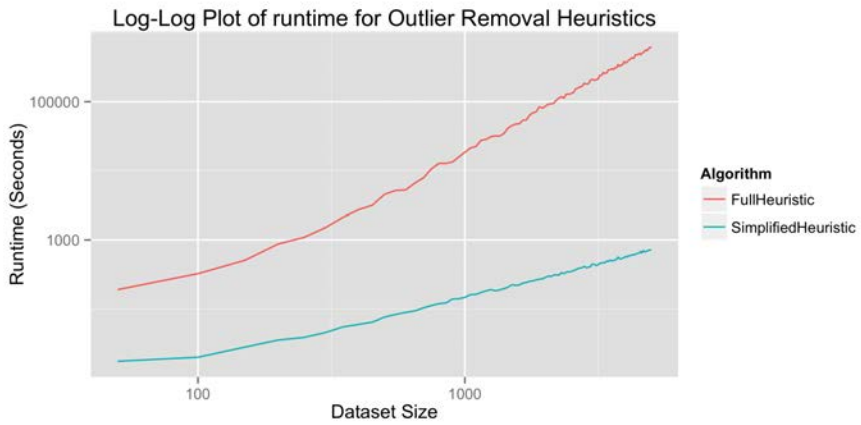


Figure 4.5: Log-log plot of the average runtime for the full heuristic and the simplified heuristic, excluding dendrogram construction.

4.6 Conclusion

Micro-benchmarking is a useful tool for the performance analysis of individual application components. In particular, micro-benchmarking represents a potential tool for use in the development of mobile systems and applications. However, micro-benchmarking features inherent complexities that must be addressed in order to produce meaningful results. The fine-grained nature of micro-benchmarking measurements reveals noise generated by the underlying hardware and operating system. This noise includes outliers that must either be removed or otherwise accounted for when processing collected data.

Existing micro-benchmarking implementations utilise only simple approaches to mitigating the effects of outliers. These approaches rarely utilise data mining techniques. This reduces the quality of the information produced by these tools. There is a lack of mobile micro-benchmarking tools that provide a comprehensive approach to outlier removal.

In this chapter, I proposed a heuristic for the automated removal of outliers from mobile micro-benchmarking datasets. The design of this heuristic was informed by the observed characteristics of micro-benchmarking datasets collected from a range of mobile devices. I then developed a simplified version of the outlier removal heuristic for use on mobile devices. Finally, I performed empirical evaluation of both the effectiveness and efficiency of the proposed heuristics.

Evaluation results demonstrate that both the full and simplified heuristic provide acceptable outlier removal effectiveness. The simplified heuristic operates in log-linear time, making it suitable for use on resource-constrained mobile devices. This makes it feasible to perform outlier removal on-device prior to exporting the collected data. The inclusion of automated outlier removal without the need for post-processing on another device represents an enhancement to the usefulness of mobile micro-benchmarking tools.

The effectiveness of the simplified heuristic demonstrates that the relationship between the characteristics of the underlying input data and the ideal cut height can indeed be captured and expressed in an easily computable form. This represents an intriguing avenue for further research. In future, I intend to collect

micro-benchmarking datasets from a wider variety of both mobile and desktop hardware. I am interested to explore alternative forms of simplification than the curve-fitting approach presented here, and determine if an overarching framework can be developed that facilitates the automated removal of outliers from micro-benchmarking data across all platforms.

Chapter 5

GPU-accelerated hierarchical clustering

The automated outlier removal technique proposed in Chapter 4 utilises the results of hierarchical clustering to remove outliers from micro-benchmarking data. The heuristic used to cut the dendrogram produced by hierarchical clustering is computationally inexpensive, making it suitable for use on resource-constrained mobile devices. However, the cost of performing hierarchical clustering itself dominates the time required to perform outlier removal. This chapter addresses this limitation by proposing a novel parallel algorithm for performing hierarchical clustering of single-dimensional data. The proposed algorithm makes use of General-Purpose Graphics Processing Unit (GPGPU) technologies to achieve massive parallelism on commodity consumer hardware. The unique characteristics of single-dimensional data are exploited to maximise the level of parallelism, performing more merges in parallel than any existing parallel implementation of hierarchical clustering.

After describing the implementation of the proposed algorithm, I validate its correctness and efficiency against the classical serial algorithm. Results demonstrate that the clusterings produced by the proposed parallel algorithm are equivalent to the clusterings produced by the classical algorithm when cutting the resulting dendrograms at every height. Benchmarking results demonstrate that the proposed parallel algorithm achieves a significant speed increase when compared to the classical serial algorithm, allowing massive single-dimensional datasets to be clustered very quickly.

The content from this chapter has been submitted for publication as:

- *Rehn, A., Possemiers, A., & Holdsworth, J. Efficient hierarchical clustering for single-dimensional data using CUDA. Submitted to* 2017 Pacific Asia Conference on Information Systems (PACIS).

5.1 Introduction

Hierarchical clustering is a widely used clustering technique, which is designed to identify a hierarchy of nested clusters within a dataset of an arbitrary number of dimensions. The most commonly utilised form of hierarchical clustering is the agglomerative, or "bottom-up" approach. In this approach, each point in a dataset is initially assigned to its own cluster, and clusters are successively agglomerated until all clusters have been merged into one. The classical algorithm for performing agglomerative clustering is extremely slow for large datasets, featuring $O(n^3)$ time complexity and $O(n^2)$ space complexity. A great deal of research has been undertaken to devise more efficient algorithms, and various approaches have been proposed that achieve optimality whilst performing all operations sequentially (Day & Edelsbrunner, 1984; Defays, 1977; Murtagh, 1985; Sibson, 1973).

In recent years, there has been a growing body of work focussed on improving the efficiency of hierarchical clustering through the use of parallelism. A number of parallel approaches have been proposed that utilise standard distributed computing architectures to spread computation over a cluster of compute nodes (Cathey, Jensen, Beitzel, Frieder & Grossman, 2007; Dash, Petrutiu & Scheuermann, 2007; Du & Lin, 2005; Feng, Zhou & Shen, 2007). More recent approaches have explored the use of parallel acceleration through General-Purpose Graphics Processing Unit (GPGPU) computing technologies (Malhat & El-Sisi, 2015; S. A. Shalom & Dash, 2013; S. A. A. Shalom, Dash & Tue, 2010; Q. Zhang & Zhang, 2006). GPGPU allows graphics rendering hardware to be utilised for general computational tasks. Due to the highly parallel nature of GPU architectures, GPGPU allows for massive levels of parallelism whilst using commonly available consumer hardware. This offers a distinct advantage over the traditional distributed computing approach, which requires a far greater investment in hardware.

Although existing GPGPU implementations of hierarchical clustering perform a great deal of processing in parallel, they typically still perform merges sequentially (Kohlhoff, Sosnick, Hsu, Pande & Altman, 2011; S. A. A. Shalom et al., 2010). The sequential merging of clusters represents a bottleneck that can significantly reduce the speed of a parallel implementation, as per Amdahl's Law (Hill & Marty, 2008). I am aware of only one existing GPGPU implementation that facilitates parallel merging, that proposed by S. A. Shalom and Dash (2013), which partitions data into a set of cells and performs merges in each cell independent of all other cells. However, merges within a given cell are still performed in sequence, and so maximum parallelism of merges cannot be guaranteed. I refer to the degree to which merges can be performed in parallel as the level of *merge parallelism*. To maximise the speed-up achieved by parallel clustering, an algorithm is needed that maximises the level of merge parallelism.

In this chapter, I propose a novel GPGPU-accelerated parallel algorithm for performing hierarchical clustering of one-dimensional data. The proposed algorithm takes advantage of the unique characteristics of one-dimensional data to maximise the level of merge parallelism, whilst ensuring that the resulting clustering remains equivalent to that produced by the classical algorithm. The proposed algorithm also features significantly reduced computational complexity and memory use when compared to the classical algorithm, and reduced space complexity when compared to the existing GPGPU parallel merge algorithm.

The contributions of this chapter are:

- I propose a novel GPGPU-based approach for performing hierarchical clustering with the maximum level of merge parallelism. The unique characteristics of single-dimensional data are exploited to achieve a worst-case time complexity of $O(\frac{n^2}{t})$, where t is the number of threads that the GPU can run simultaneously, and a space complexity of $O(n)$.
- I implement my proposed algorithm using the NVIDIA Compute Unified Device Architecture (CUDA) language, and describe the specific CUDA functionality utilised.
- I validate the results produced by the proposed algorithm against those produced by the classical algorithm, and demonstrate that the clusterings are equivalent when cutting the hierarchical trees at all heights.
- I benchmark the speed of the proposed algorithm against a widely-used implementation of the classical algorithm and demonstrate a significant speed-up as the size of the dataset being clustered increases.

The rest of this chapter is structured as follows. First, I provide background details on the classical hierarchical clustering algorithm, GPGPU technologies, and existing parallel implementations of hierarchical clustering. Next, I describe my proposed algorithm and its implementation in the CUDA language. Then, I validate both the efficiency and correctness of my proposed algorithm against the classical implementation. Finally, I discuss future research directions to extend the work presented in this chapter.

5.2 Background

5.2.1 Hierarchical Agglomerative Clustering (HAC)

Hierarchical Agglomerative Clustering (HAC) is a clustering approach that was first proposed for clustering taxonomies of genetic data in microbiology (Sneath, 1957), and now enjoys wide use in a variety of areas such as document retrieval (Aboutabl & Elsayed, 2011), bioinformatics (Wilson, Dai, Jakupovic, Watson & Meng, 2007), and data mining (B. Wang, Ding & Rahal, 2008). The distinguishing characteristic of hierarchical clustering is that it produces a binary tree of clusters instead of a single set of partitions. This tree, known as a dendrogram, contains N leaf nodes, representing each point in the input dataset, and $N-1$ interior nodes, representing the successive merges of the clusters. As stated in Chapter 4, each interior node is labelled with a value called the *height*, which represents the distance at which that node's child clusters were merged. The dendrogram can then be "cut" at a given height to extract the clustering at that particular level of the nested hierarchy.

HAC is a paramaterised algorithm that accepts two parameters:

1. A *distance metric* that defines the dissimilarity between any two points in the dataset that is being clustered.

2. A *linkage criteria* that defines the dissimilarity of two clusters of points, as a function of the pairwise distances between the points in those clusters. Common linkage criteria include *single-linkage*, defined as the minimum pairwise distance in the set of distances between the points in two clusters, *complete-linkage*, defined as the maximum pairwise distance in the set of distances between the points in two clusters, *average-linkage*, defined as the mean of the the set of distances between the points in two clusters, and *centroid-linkage*, defined as the distance between the centroids of the two clusters.

Given a set of values, x, the classical implementation of the algorithm is as follows:

1. Assign each value in x its own cluster.
2. Compute an $N \times N$ matrix of dissimilarity values, known as the *distance matrix*, where $N = |x|$, and the value at position $[i, j]$ is the distance between the values x_i and x_j, as defined by the selected distance metric. The values along the diagonal of the distance matrix represent the distance between each cluster and itself, and are therefore zero.
3. Locate the smallest value in the distance matrix (ignoring the values along the diagonal), and merge the clusters identified by the position of that distance value in the matrix.
4. Where i and j are the indices of the merged clusters, remove rows i and j, and columns i and j, from the distance matrix. Add a new row and a new column to the distance matrix, and populate it with the distances between the newly-formed cluster and all other clusters, as defined by the linkage criteria.
5. Repeat steps 3 and 4 until all clusters have been merged into a single cluster.

The classical implementation of the algorithm features $O(n^3)$ time complexity, and $O(n^2)$ space complexity for storing the distance matrix. There are a number of implementations featuring quadratic time complexity that target specific linkage metrics. For the single-linkage criteria, the optimal algorithm SLINK (Sibson, 1973) produces comparable results to the classical implementation. A similar optimal algorithm, CLINK (Defays, 1977), exists for the complete-linkage criteria, but produces results which are inferior to those produced by the classical implementation (El-Hamdouchi & Willett, 1989; Rasmussen, 1992). Similar algorithms exist for average-linkage (Murtagh, 1985) and centroid-linkage (Day & Edelsbrunner, 1984). Alternative implementations have also been proposed that improve efficiency through the use of sampling (Guha, Rastogi & Shim, 1998), summarisation (T. Zhang, Ramakrishnan & Livny, 1996) and other heuristic measures (Dash, Tan & Liu, 2001).

5.2.2 General Purpose GPU (GPGPU) languages

Prior to the popularisation of GPGPU technologies, the only way that GPU architectures could be used for performing general computational tasks was to implement code in the form of a *shader*. Shaders are code that is executed by the programmable stages of the graphics rendering pipeline, and were not originally designed for general computation. Newer graphics libraries support a special type of shader known as a

compute shader (Ni, 2009), but these are still restricted when compared to dedicated GPGPU programming languages.

The two most popular GPGPU programming languages are the Khronos Group's OpenCL (Khronos Group, 2016) and NVIDIA's Compute Unified Device Architecture (CUDA) (NVIDIA Corporation, 2016b). OpenCL is an open standard that runs on graphics hardware from multiple vendors, whilst CUDA is a proprietary technology that only runs on graphics hardware manufactured by NVIDIA. However, the older versions of OpenCL supported by most vendors at the time of writing only support the C programming language. In addition, there is relatively little library support provided by the Khronos Group for OpenCL. In contrast, CUDA supports the far more flexible C++ programming language, and ships with a robust standard library known as *thrust* (NVIDIA Corporation, 2016c). The thrust library provides optimised parallel implementations of common algorithms, significantly reducing programmer burden. It is for this reason that I have selected CUDA for use in this research.

Under the CUDA programming model, functions that are written to be executed in parallel are referred to as *kernels*. The execution of a kernel is represented by a 1, 2, or 3-dimensional grid of *blocks*, which in turn contain a series of *warps*. This model reflects the underlying hardware implementation of NVIDIA GPUs. On current-generation GPUs at the time of writing, the maximum number of threads within a block is 1024, and the number of threads in a warp is 32 (NVIDIA Corporation, 2016a). Blocks are mapped to the graphics card's computational hardware by the CUDA scheduler, and executed in parallel. The number of blocks that can be executed concurrently is determined by the number of Streaming Multiprocessor (SM) units that the GPU features.

There are a number of factors that must be taken into consideration when writing CUDA code to achieve maximum performance. The first is the CUDA memory model. All of the threads in a single block have access to fast cache memory known as "shared" memory, and also to GPU main memory, which is referred to as "global" memory. Memory accesses to shared memory are faster than accesses to global memory, and it may be necessary to transfer data between the two to maximise performance. In addition, contiguous memory accesses are faster than distant memory accesses (NVIDIA Corporation, 2016a). The second factor that must be taken into consideration is the Single Instruction Multiple Thread (SIMT) architecture of CUDA compute cores, which is akin to the traditional Single Instruction Multiple Data (SIMD) architecture model (NVIDIA Corporation, 2016a). Due to the nature of this architecture, all threads within a warp must execute the same code. If the code being executed contains branches and threads within a warp take different paths, the CUDA scheduler must first execute all threads that take one path, and then proceed to execute the threads that took the other path. This scenario is known as *thread divergence*, which results in a significant reduction in parallelism (NVIDIA Corporation, 2016a).

More general concerns that must be taken into account when implementing parallel algorithms include synchronisation and maximisation of parallelism. Because the order in which threads are executed is not known, the same data dependency and synchronisation issues from the traditional multi-threaded programming model must considered. Careful use of synchronisation is important to avoid a reduction in parallelism. The algorithm itself must also be designed to perform as much processing in parallel as possible.

As Amdahl's Law states, the speed improvements introduced by parallelism are limited by the proportion of the processing that is actually performed in parallel (Hill & Marty, 2008). Serial processing must be minimised as much as possible for the benefits of the parallel portion of the code to be maximised.

5.2.3 Parallel implementations of HAC

There exist a number of approaches for implementing HAC in parallel on various target architectures. Table 5.1 summarises a number of these approaches.

Table 5.1
A sample of existing approaches for implementing hierarchical clustering in parallel.

Implementation(s)	Linkage metrics supported	Implementation details
Li, Li, Xiao and Yang (2007); Olson (1995); Rajasekaran (2005)	Single, complete, average, centroid, median, minimum variance	Targeting PRAM architectures
Arumugavelu and Ranganathan (1996)	Single, complete	Targeting SIMD architectures
Wu, Horng and Tsai (2000)	Single	Targets AROB architectures
Hadjidoukas and Amsaleg (2008); Hendrix, Patwary, Agrawal, k. Liao and Choudhary (2012)	Single	OpenMP
Cathey et al. (2007); Chan, Gao and Rau-Chaplin (2005); Dash et al. (2007); Du and Lin (2004, 2005); Feng et al. (2007); Hendrix et al. (2013); B. Wang et al. (2008); B. Wang and Dong (2008)	Single, centroid	MPI
Jin et al. (2015)	Single	Spark
Jeon and Yoon (2015); Tantono (2015)	Single	Multi-threaded
Q. Zhang and Zhang (2006)	Single, complete, average	GLSL Shaders
Malhat and El-Sisi (2015)	Ward	OpenCL
D. J. Chang, Desoky, Ouyang and Rouchka (2009); D.-J. Chang, Kantardzic and Ouyang (2009); Kohlhoff et al. (2011); S. A. Shalom and Dash (2013); S. A. A. Shalom et al. (2010); S. A. A. Shalom, Dash, Tue and Wilson (2009); Wilson et al. (2007)	Single, complete	CUDA

Several older implementations target abstract machine architectures such as PRAM, SIMD, and AROB, and have not been implemented for real hardware (Li et al., 2007; Olson, 1995; Rajasekaran, 2005; Wu et al., 2000). Numerous implementations utilise libraries for distributed computing, and are designed to run on clusters of compute nodes (Cathey et al., 2007; Chan et al., 2005; Dash et al., 2007; Du & Lin, 2004, 2005; Feng et al., 2007; Hendrix et al., 2013; Jin et al., 2015; B. Wang et al., 2008; B. Wang & Dong, 2008). A smaller number of implementations are designed for CPU-level parallelism on a single device (Hadjidoukas

& Amsaleg, 2008; Hendrix et al., 2012; Jeon & Yoon, 2015; Tantono, 2015). Most recent implementations are designed to run on GPUs (Malhat & El-Sisi, 2015; S. A. Shalom & Dash, 2013; S. A. A. Shalom et al., 2010; Q. Zhang & Zhang, 2006), to exploit the immense benefits GPGPU technologies present by enabling massive parallelism on commodity hardware.

Most of the GPU-based implementations of HAC simply aim to accelerate individual phases of the clustering process (Malhat & El-Sisi, 2015; S. A. A. Shalom et al., 2010, 2009), and do not fundamentally modify the clustering algorithm itself to benefit from parallelism in the way that some previous approaches do. The implementation proposed by S. A. Shalom and Dash (2013) achieves merge parallelism by partitioning data into cells and performing merges with cell-level parallelism. However, because the merges within each cell are still performed in serial, merge parallelism is not maximised. In addition, the memory usage of GPGPU approaches, including the parallel merging algorithm, is not significantly reduced over a CPU-based implementation. As a result, memory usage can still become quite large when clustering massive datasets, thus incurring additional overheads when transferring large quantities of data between the system memory and GPU memory. Although these problems have proven complex to solve for clustering data of an arbitrary number of dimensions, the unique characteristics of one-dimensional data can be exploited to address these limitations, whilst maximising merge parallelism and still ensuring results that are equivalent to the classical algorithm.

5.3 Implementation

5.3.1 Benefits of single-dimensional data

Single-dimensional data features a number of characteristics that can be exploited in order to perform clustering efficiently. Foremost among these characteristics is that if a dataset is sorted in ascending order, it becomes a set of monotonically increasing values, totally ordered under $\leq$. Inherent in this definition is the guarantee that the distance between the values at positions i and $i+2$ cannot be less than the distance between the values at positions i and $i+1$.

The first consequence of this property is the closest neighbour of each point is already known, based purely on the positions within the sorted array. It is therefore no longer necessary to maintain a two-dimensional distance matrix containing the distances between all clusters. Only the distances between immediate neighbours are needed. This information can be stored in a single-dimensional array, immediately reducing space complexity from $O(n^2)$ to $O(n)$. As a result, searching the distances array to find the global minimum can be achieved in $O(n)$ time, compared to $O(n^2)$ required to search a two-dimensional distance matrix. A number of existing implementations utilise similar data structures containing immediate neighbour distances (Cathey et al., 2007; Jeon & Yoon, 2015; Olson, 1995). However, these implementations utilise such structures in addition to the two-dimensional distance matrix, and do not eliminate the distance matrix itself.

The second consequence of the monotonically increasing property is that the bounds of a given cluster can be determined by simply storing the minimum and maximum values that fall within that cluster. This has

particular importance to the single-linkage and complete-linkage metrics. For both of these metrics, the bounds of two clusters can be used to calculate the distance between them. The calculation of each new distance requires only a single subtraction operation, as opposed to the slightly more expensive operation of finding the minimum or maximum of the existing distances. More importantly, because each thread can calculate new distances without reading the existing contents of the distances array, the array can be safely updated in parallel after merges have occurred without any chance of race conditions. This makes it possible to perform merges in parallel.

In the section that follows, I describe my algorithm for clustering one-dimensional data in parallel. I focus exclusively on supporting the single-linkage and complete-linkage metrics, due to the benefits that can be exploited when utilising these linkage metrics for one-dimensional data.

5.3.2 Algorithm

Algorithm 5.1 High-level overview of my algorithm, prior to adding merge collision detection or duplicate preprocessing.

Input: V - the set of values to be clustered
Output: the list of merges
1: $clusters \leftarrow \emptyset$
2: $distances \leftarrow \emptyset$
3: $merges \leftarrow \emptyset$
4: $N \leftarrow |V|$
5: copy V from host memory to device memory
6: sort V
7: $clusters = V$
8: **while** $|merges| < (N-1)$ **do**
9: $distances = \text{CALCULATEDISTANCES}(clusters)$
10: $minDist = \text{MIN}(distances)$
11: (clusters, merges) = $\text{PERFORMMERGES}(clusters, merges, distances, minDist)$
12: $clusters = \text{COMPACTARRAY}(clusters)$
13: **end while**
14: copy *merges* from device memory to host memory
15: **return** *merges*

Algorithm 5.1 depicts the pseudocode for the high-level logic of my algorithm, prior to the addition of behaviour to detect parallel merge conditions or pre-process duplicate values. The algorithm operates as follows:

1. First, an array of length $N-1$ is allocated in GPU memory to hold the list of merges, where N is the number of values that is being clustered. Each item in the list contains the indices of the two merged clusters and the distance value at which they were merged. This is one possible representation of a dendrogram, whereby each merge in the list represents an internal node in the tree. A counter variable is also created to keep track of the number of merges that have been performed, with an initial value of zero.

2. Next, the set of values that is being clustered is copied from system memory to GPU memory, and then sorted on the GPU in parallel. For this purpose, I utilise the CUDA *thrust::sort()* function, which is

implemented as a parallel radix sort and has a time complexity of $O(n)$ (Merrill & Grimshaw, 2011).

3. The initial array of clusters is then built from the sorted data. Each cluster stores its lower and upper bound, as well as a flag to indicate if the cluster has been merged. One cluster is created for each point in the sorted dataset, with both its lower and upper bound set to the value of the point that it contains. The flag to indicate merge status is set to FALSE.

4. At this point, the main loop commences its first iteration. The pairwise distances between all immediately neighbouring clusters are computed in parallel, populating the distance array.

5. Once the distances have been computed, the minimum distance value is determined using the CUDA *thrust::min_element()* function, which is implemented as a parallel reduction algorithm and has a time complexity of $O(n)$ (Hoberock & Bell, 2016).

6. With the global minimum value available, merges can be performed in parallel. One thread is launched per cluster. The thread with index i determines if the distance between cluster i and cluster $i+1$ is equal to the global minimum. If so, the upper bound of cluster i is set to the upper bound of cluster $i+1$, and cluster $i+1$ has its merged flag set to TRUE. To ensure data consistency, a single synchronisation operation is utilised. The thread atomically increments the merge counter using the CUDA *atomicAdd()* function, which returns the original value prior to being incremented. This value is used as an index representing the position in the merges array in which the merge details should be written. The synchronisation overhead of the atomic add is minimal and allows an arbitrary number of merges to be performed in parallel, whilst preventing the merges array from becoming corrupted.

7. To complete the current loop iteration, the array of clusters is compacted. All clusters with the merge flag set to TRUE are removed from the array, which ensures that neighbouring positions in the array continue to hold neighbouring clusters. This is implemented using the CUDA *thrust::remove_if()* function, which shifts removed values to the end of the array in parallel, and returns the new length.

8. If the number of remaining clusters is still greater than one, the loop iterates again, returning to Step 4.

9. Once all points have been merged into a single cluster, the list of merges is copied from GPU memory to system memory. The retrieved list of merges represents the completed dendrogram for the clustering.

The use of parallel merges reduces the overall number of iterations required for the algorithm to complete. As a result, $N-1$ iterations becomes the worst-case scenario, which only occurs when clustering datasets in which all inter-cluster distances are unique at all stages of the clustering process. However, there are a number of considerations that must be taken into account when performing merges in parallel in order to ensure the correctness of the algorithm. These considerations are discussed in the section that follows.

5.3.3 Handling merge collisions

Consider a scenario in which there exist three neighbouring clusters, i, j, and k, and the distances between clusters are equal:

$$d(i,j) = d(j,k)$$

When merging in parallel, the thread for cluster i will attempt to merge $i+j$, whilst the thread for cluster j will simultaneously attempt to merge $j+k$. The combination of these clusterings is invalid, since cluster j ceases to exist after it is first merged, and therefore cannot be merged a second time. I define this situation as a *collision*. Collisions must be prevented in order to ensure correct output.

A naive collision detection algorithm that directly implements the definition of a merge collision is as follows: in the scenario where the thread processing the cluster with index i wishes to merge clusters i and $i+1$, a collision can be avoided by simply checking if cluster $i-1$ will merge with cluster i, and cancel the merge of i and $i+1$ accordingly. However, this solution can result in a serious reduction in the number of merges than can be performed in parallel. Consider a scenario where there exists a set of x neighbouring clusters with identical inter-cluster distances. Based on the definition of a collision outlined above, there exists a set of $|x|-1$ contiguous collisions, which include all but the first of the clusters in the set x. The maximum number of merges that can safely be performed in parallel is $\frac{|x|}{2}$, but the naive collision detection implementation will only perform the left-most merge, treating all collisions as unmergeable. This scenario is illustrated in Figure 5.1.

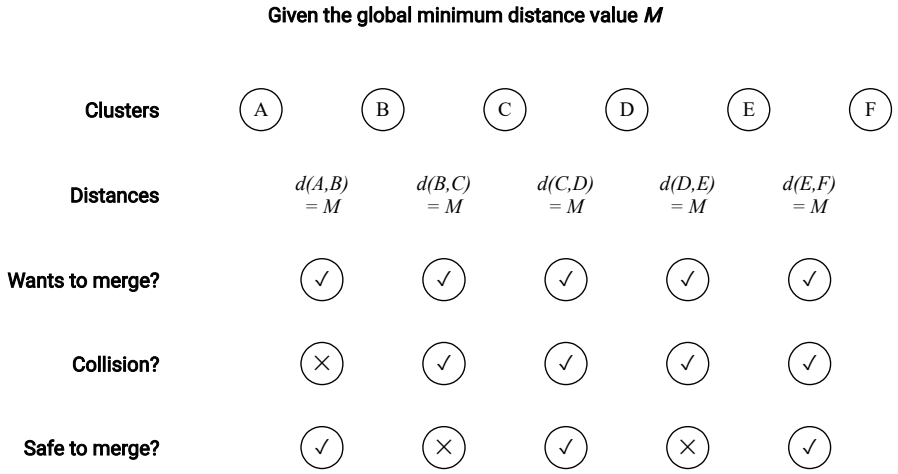


Figure 5.1: Example of a scenario where merge collisions exist. Only the merges that are not marked as collisions will be merged in parallel by the naive collision detection algorithm, whereas all of the merges marked as "safe to merge" are able to be safely merged in parallel.

To achieve the full number of parallel merges, a more sophisticated approach is required. Simply detecting that a merge is a collision is not sufficient. It is necessary to determine the offset of each collision within a set of contiguous collisions, so that every second collision within the set can be merged. Additionally, multiple sets of contiguous collisions may exist, and so it is necessary to determine the index at which each set begins, so that the offsets for the collisions within a given set can be computed relative to the starting

index for that set. My proposed collision detection algorithm is as follows:

1. Allocate an array of boolean flags with length $N-1$, and an array of integers with length $N-1$, where N is the number of clusters being considered. The array of boolean flags represents detected collisions, and its values are initialised to FALSE. The array of integers represents the indices of the first collision in each set of contiguous collisions, and its values are initialised to zero.

2. Invoke in parallel the naive collision detection algorithm described above, such that the thread with index i will set the boolean flag at array index i to indicate if a merge collision was detected preventing the merge of cluster i and $i+1$. The thread will also check if a merge collision exists that prevents the merge of cluster $i-1$ and i. If no such collision is detected, the collision at position i is the first collision in a set of contiguous collisions, and the integer value of i is written to position i in the array of integers.

3. Invoke in parallel an inclusive scan operation, using the MAX binary operator, which returns the greater of its two operands. An inclusive scan is a generalised version of the classical prefix sum operation from computer science. Whereas a prefix sum is defined to use only the addition operator, an inclusive scan can utilise any arbitrary binary operator. I utilise the CUDA *thrust::inclusive_sum()* function for this purpose, which implements an inclusive scan in parallel. The result of this step is that every position in the array of integers that corresponds to a collision contains the index of the first collision in the contiguous set of collisions to which that collision belongs. This is because the MAX operator propagates the indices to the right, filling all array positions that contain a zero value with the closest left-most non-zero value. In the scenario that a set of contiguous collisions starts at index zero, the array positions for all of the collisions within that set will remain zero, which is the correct index value.

4. When the function to perform merges is invoked in parallel, it receives both the array of collision flags and the array of indices. The thread with index i first checks that the distance between clusters i and $i+1$ is equal to the global minimum. If it is, the thread checks the value of the collision flags array at position i to determine if the merge is a collision. If it is not a merge collision, the merge proceeds. If the merge is a collision, the thread reads the index value at position i of the integer array, and subtracts that value from i to determine the offset of the current collision within its containing set of contiguous collisions. Since the offset will be zero for the first collision in a set, the offset for every second value will be an odd number. If the offset is an odd number, the merge proceeds. Otherwise, the collision stands and the merge is not performed.

An example of the collision detection algorithm being applied to a simple dataset is illustrated in Figure 5.2. Although the collision detection algorithm introduces additional processing, it facilitates the maximum level of parallel merging, thus significantly reducing the number of required iterations of the overarching clustering algorithm for datasets where identical inter-cluster distances exist. However, the correctness of the produced clusters relies on an important assumption about the dataset being clustered. I discuss this assumption, and the method for ensuring its validity, in the section that follows.

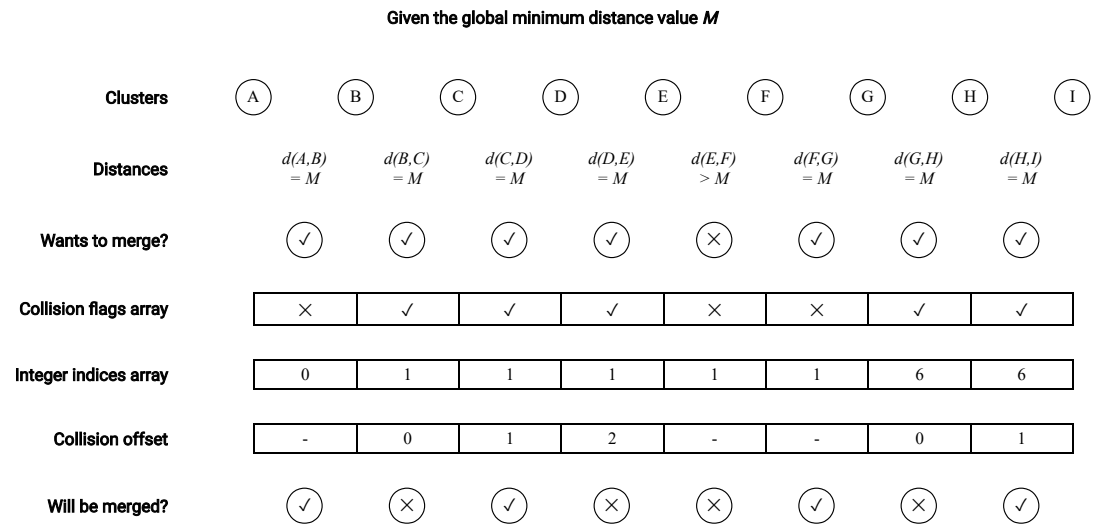


Figure 5.2: Example run of the proposed collision detection algorithm, which maximises merge parallelism.

5.3.4 Pre-merging duplicate values

Recall from the section above that, given a set x of contiguous merge collisions, the maximum number of merges that can safely be performed in parallel is $\frac{|x|}{2}$, whereby every second collision is merged. This is the largest set of parallel merges that will produce a well-formed dendrogram. However, whether or not this clustering matches the results produced by the classical algorithm depends on the data being clustered and the linkage metric being used.

Consider the scenario where there exist four clusters, i, j, k, and l, with identical inter-cluster distances. The parallel algorithm will produce the merges $i+j$ and $k+l$. If the distance between $i+j$ and k is greater than the distance between k and l, the classical algorithm will produce the same merges. However, if the distance between $i+j$ and k is less than or equal to the distance between k and l, the classical algorithm will produce the merge $i+j$ followed by the merge $(i+j)+k$. This scenario is illustrated in Figure 5.3.

For the single-linkage metric, these clusterings are equivalent. Under single-linkage, $d(i, j+k) = d(i, j)$ and $d(j+k, l) = d(k, l)$, which means that the immediate neighbours of a newly merged cluster are the remaining (non-merged) neighbours of the two clusters that were merged (Olson, 1995). As a consequence, the order of the merges in this scenario does not alter any of the distances involved, and so the final dendrogram contains the same number of interior nodes with the same heights. Although the topology of the trees produced by the classical and parallel algorithms are not identical, the clustering produced by cutting the dendrogram at $d(i, j)$ will be the same.

However, the same does not hold true for the complete-linkage metric. The distance between a newly merged cluster and its immediate neighbours changes as a result of the merge, and so the order of merges in the scenario above does matter. When utilising the complete-linkage metric, the parallel algorithm will fail to produce merges that are equivalent to the classical algorithm, for the clusters i, j, k, and l, when the following conditions are satisfied:

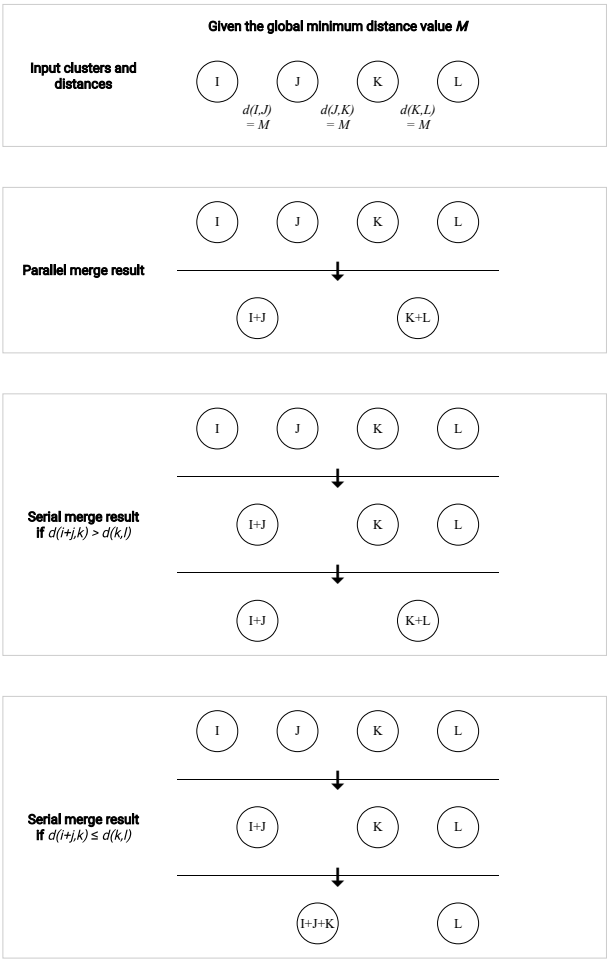


Figure 5.3: An example of a scenario where the clustering decisions made by the parallel algorithm can differ from those made by the classical serial algorithm, depending on the properties of the points being clustered.

$$d(i,j) = d(k,l)$$

$$d(i+j,k) \leq d(k,l)$$

Under complete-linkage, the relevant distances between the clusters are defined in terms of the lower and upper bounds of the clusters:

$$d(i,j) = j_{upper} - i_{lower}$$

$$d(k,l) = l_{upper} - k_{lower}$$

$$d(i+j,k) = k_{upper} - i_{lower}$$

In order to satisfy the conditions $d(i,j) = d(k,l)$ and $d(i+j,k) \leq d(k,l)$, the following must hold:

$$k_{upper} - i_{lower} \leq j_{upper} - i_{lower}$$

Which can be simplified to become:

$$k_{upper} \leq j_{upper}$$

Since the ordered nature of the data prevents $k_{upper} < j_{upper}$ from ever holding true, this can be simplified further to become $k_{upper} = j_{upper}$. Given that $j_{upper} \leq k_{lower} \leq k_{upper}$, it is therefore known that for the simplified condition to hold true, $k_{lower} = k_{upper} = j_{upper}$ must also hold true. As such, the parallel algorithm can only fail to produce the correct clustering when a cluster exists with identical upper and lower bounds, that are equal to the upper bound of the cluster's immediate neighbour to the left. This situation can only arise during the first loop iteration of the parallel algorithm, and only if the dataset being clustered contains duplicate values. In order to prevent this from occurring, it is necessary to add a pre-processing step to merge all identical values at the beginning of the algorithm, prior to the first loop iteration.

To pre-process all duplicate values, a custom merge function is launched in parallel, immediately after the initial sort of the input dataset. All neighbouring points with a distance value of zero are merged, without checking for collisions. Collisions are resolved by sorting the list of merges in descending order of the index of the left-hand cluster in each merge. This sort utilises the same CUDA *thrust::sort()* parallel sorting function used for the initial sort of the input dataset. Sorting the list of merges in descending order of left-hand cluster index enforces a right-to-left ordering of merges. This is necessary because merges eliminate the right-hand cluster by merging it into the left-hand cluster, and a right-to-left ordering ensures that resulting dendrogram nodes are well-formed. This concept is illustrated in Figure 5.4.

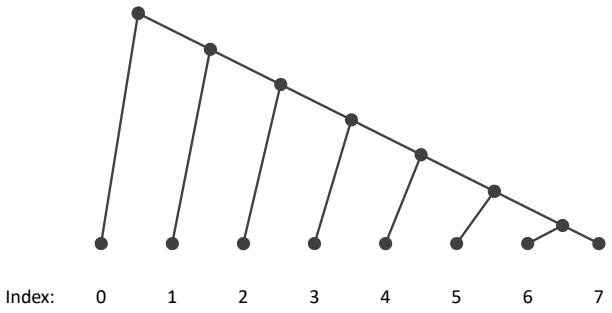
Pre-processing duplicate values ensures the validity of the clustering output under the complete-linkage metric because it eliminates the only situation which can satisfy the conditions necessary to generate incorrect clustering decisions. In addition, pre-processing can provide speed benefits when clustering datasets with large numbers of duplicate values, since a large number of merges will be performed in parallel prior to the beginning of the loop, resulting in a reduction in the total number of required loop iterations.

5.4 Evaluation

To evaluate both the correctness and efficiency of my proposed parallel algorithm, I compare it to the classical hierarchical clustering implementation from the R statistical programming language. The combination of

LHS Cluster	RHS Cluster	Distance
6	7	0
5	6	0
4	5	0
3	4	0
2	3	0
1	2	0
0	1	0

(a)



(b)

Figure 5.4: Example results of the duplicate value pre-processing step. (a) is the sorted list of merges, and (b) is the resulting dendrogram.

the R *dist()* and *hclust()* functions represents a mature, stable, and well-optimised implementation of the classical serial version of the algorithm, featuring $O(n^3)$ time complexity and $O(n^2)$ space complexity, and supporting datasets with an arbitrary number of dimensions. The R implementation is also widely used, making it an ideal representative of the classical serial algorithm against which to compare the proposed parallel algorithm.

All experiments were performed on a desktop machine running Ubuntu 16.10 Linux, featuring an Intel Core i5-6400 CPU, 16GB of system memory, and 60GB of swap space. Three GPUs were used for performing benchmarks: an NVIDIA GeForce GTX 570, an NVIDIA GeForce GTX 970, and an NVIDIA GeForce GTX 1080. For each GPU, the CUDA code was recompiled using the compiler flags corresponding to the highest CUDA Compute Level supported by that GPU. The version of the NVIDIA graphics driver installed on the test system was release 367.57.

For all experiments, three types of test datasets are utilised:

Gaussian. Values randomly sampled from the Gaussian (normal) distribution.

Poisson. Values randomly sampled from the Poisson distribution.

Random. "True random" values retrieved from the *random.org* web service, which utilises atmospheric data to generate high quality random byte streams (Haahr, 2016).

For each dataset type, a series of datasets of increasing sizes are generated, ranging in size from 100 points to 100,000 points. The full range of generated datasets is utilised when benchmarking the performance of the parallel version of the clustering algorithm. However, the R *hclust()* function enforces a maximum limit of 65536 points (R Core Team, 2016). Additionally, when attempting to perform clustering of datasets of 50,000 points and larger, I found that the R implementation would crash on my test machine due to insufficient system memory. As such, when benchmarking the performance of the serial version of the clustering algorithm, and when verifying the output of the parallel algorithm against that of the serial algorithm, the dataset size is capped at 45,000.

5.4.1 Validation of correctness

To validate the correctness of my parallel clustering algorithm, I compare the dendrograms produced by the parallel algorithm to those produced by the R *hclust()* function. To ensure that there are no differences caused by the ordering of the input data, each input dataset is sorted in R prior to performing clustering with the classical algorithm. This sorting step is only performed in the validation experiment and is not included in the performance benchmarks of the classical algorithm.

There are numerous measures of dendrogram similarity that focus on tree topology (Baker, 1974; Robinson & Foulds, 1981; Scornavacca, Zickmann & Huson, 2011). However, I am more interested in the clusterings produced by cutting the dendrograms than the similarity of their topologies. The clusterings produced by cutting a dendrogram are a more meaningful representation of the clustering result than the specific order of the interior nodes of the dendrogram, particularly for datasets where duplicate values exist. To verify the clustering similarity, I first extract the set of unique height values from the interior nodes of both dendrograms. If the two sets of extracted height values do not match exactly, I consider the output of the parallel algorithm to be incorrect. Then, for each height in the set of unique heights, I cut both dendrograms at that height and extract the resulting clusterings. I compare these clusterings using the Fowlkes-Mallows Index (Fowlkes & Mallows, 1983), a well-known clustering similarity metric. The Fowlkes-Mallows Index for two sets of clusterings is a value in the range of zero to one. A value of zero indicates maximum dissimilarity between the two clusterings, whilst a value of exactly one indicates that the two clusterings are identical. If the clusterings produced by cutting the two dendrograms at any given height yield a Fowlkes-Mallows Index value of less than exactly one, I consider the output of the parallel algorithm to be incorrect.

For all dataset sizes of all three dataset types, and for both single-linkage and complete-linkage metrics, the dendrograms produced by the parallel algorithm successfully pass validation. There is no room for error in the validation conditions I define. For a pair of dendrograms to pass validation, they must be exactly equivalent in all of their clusterings, save for the specific details of their tree topology. The 100% validation pass rate demonstrates that the results of my parallel clustering algorithm are exactly equivalent to the results of the serial algorithm implementation of R's *hclust()* function. As the benchmarking results discussed in the next section demonstrate, this equivalent output is produced by the parallel algorithm in a fraction of the time taken by the serial algorithm, especially as the size of the dataset being clustered increases.

5.4.2 Performance benchmarking

To validate the efficiency of the proposed parallel clustering algorithm, I benchmark the runtime of the clustering process for both the parallel algorithm and the classical serial algorithm. The benchmarked runtime includes only the clustering process itself and not other processing such as reading the input dataset from a file on disk. For the classical algorithm, the benchmarked runtime includes both the R *dist()* and *hclust()* function calls, which construct the distance matrix and perform clustering using that distance matrix, respectively. Each dataset is benchmarked 10 times with each algorithm implementation and each linkage metric, and the mean of each set of 10 runs is used as the reported value. When benchmarking the classical

algorithm, only dataset sizes up to 45,000 are used, due to system memory constraints. When benchmarking the parallel algorithm, the full range of dataset sizes up to 100,000 is used.

Recall that the classical algorithm implemented by R features a time complexity of $O(n^3)$. Thanks to the use of a single-dimensional distances array, all operations within the loop of the parallel algorithm operate in $O(\frac{n}{t})$ time, where t is the number of threads that can be run in parallel on the GPU. The loop itself has a worst-case iteration count of $n-1$, and so the worst-case time complexity of the parallel algorithm is $O(\frac{n^2}{t})$. In situations where $t \geq n$, the worst-case runtime effectively becomes $O(n)$. In addition, the number of loop iterations is dependent on the values in the dataset being clustered, and can be significantly less than $n-1$ in situations where a large number of duplicate values or values with equal inter-cluster distances are present. The pseudocode for the complete version of the parallel algorithm, with merge collision detection and duplicate value pre-processing steps, is listed in Algorithm 5.2.

Algorithm 5.2 The complete algorithm with merge collision detection and duplicate value pre-processing functionality added. The comments on the right indicate the worst-case time complexity of each operation in the algorithm, where t is the number of threads that the GPU can run in parallel.

Input: V - the set of values to be clustered
Output: the list of merges

1: $clusters \leftarrow \emptyset$
2: $distances \leftarrow \emptyset$
3: $merges \leftarrow \emptyset$
4: $N \leftarrow |V|$
5: copy V from host memory to device memory ▷ $O(1)$
6: sort V ▷ $O(\frac{n}{t})$
7: $clusters = V$
8: $distances =$ CALCULATEDISTANCES($clusters$) ▷ $O(\frac{n}{t})$
9: (clusters, merges) = MERGEDUPLICATES($clusters, merges, distances$) ▷ $O(\frac{n}{t})$
10: sort *merges* in descending order of LHS cluster index ▷ $O(\frac{n}{t})$
11: $clusters =$ COMPACTARRAY($clusters$) ▷ $O(\frac{n}{t})$
12: **while** $|merges| < (N-1)$ **do** ▷ $O(n)$
13: $distances =$ CALCULATEDISTANCES($clusters$) ▷ $O(\frac{n}{t})$
14: $minDist =$ MIN($distances$) ▷ $O(\frac{n}{t})$
15: (collisions, indices) = DETECTMERGECOLLISIONS($clusters, distances, minDist$) ▷ $O(\frac{n}{t})$
16: (clusters, merges) = PERFORMMERGES($clusters, merges, distances, minDist, collisions, indices$) ▷ $O(\frac{n}{t})$
17: $clusters =$ COMPACTARRAY($clusters$) ▷ $O(\frac{n}{t})$
18: **end while**
19: copy *merges* from device memory to host memory ▷ $O(1)$
20: **return** *merges*

Figure 5.5 depicts the runtimes for the classical clustering algorithm running on the CPU, and the parallel algorithm running on each of the three GPUs in turn, for all dataset types and linkage metrics up to a dataset size of 45,000. At the maximum dataset size, the runtime of the classical algorithm reaches values of between 60 and 90 seconds, whilst the runtime for the parallel algorithm on any of the GPUs remains below 1 second. The runtime of the classical algorithm grows at such a rate that the scale of the graph obscures the differences between the three GPU models. Figure 5.6 depicts the runtimes for the parallel algorithm only, providing a clearer picture of the differences between the three GPU models.

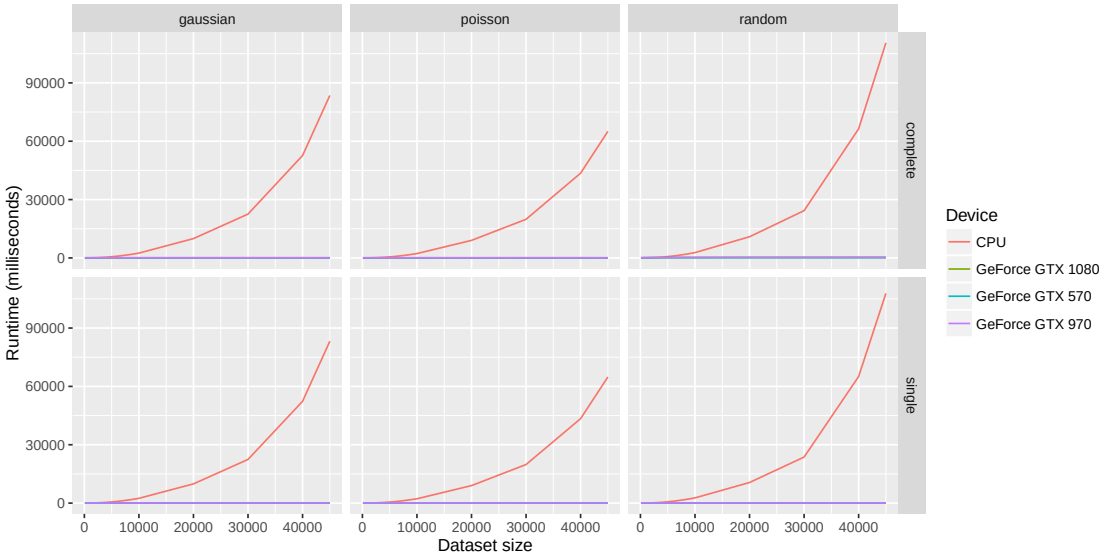


Figure 5.5: Benchmarking results for the CPU and GPU implementations of both complete linkage and single linkage, across the three dataset types.

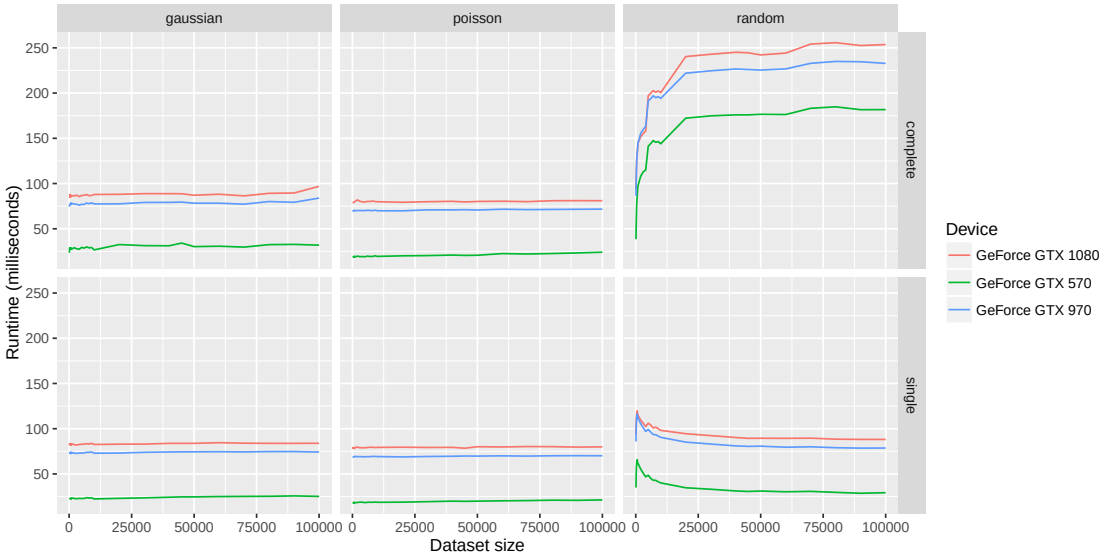


Figure 5.6: Benchmarking results for the GPU implementations of both complete linkage and single linkage, across the three dataset types.

The benchmark results for the three GPU models display an unexpected trend. Across all dataset sizes and types, and for both single-linkage and complete-linkage, the older GeForce GTX 570 outperforms the significantly newer GeForce GTX 970 and GeForce GTX 1080. In addition to this, the GeForce GTX 970 slightly outperforms the newer GeForce GTX 1080. This runs counter to the expected result, since the GeForce GTX 1080 has by far the highest clock speed of the three devices and is capable of running the greatest number of threads simultaneously. The device characteristics of the three GPUs, as reported by the CUDA *cudaGetDeviceProperties()* function, are listed in Table 5.2.

The most likely explanation of this trend is that the CUDA driver software features more mature support for older GPU models, which benefit from years' worth of accumulated optimisations. It is entirely possible that

Table 5.2
Device characteristics for the three GPU models used for benchmarking, as reported by the CUDA *cudaGetDeviceProperties()* function. The value reported for the maximum number of threads is the number of Streaming Multiprocessor (SM) units the device has, multiplied by the maximum number of concurrent threads per SM.

Device model	Clock speed	Maximum number of concurrent threads
GeForce GTX 1080	1.8095 GHz	40960
GeForce GTX 970	1.2405 GHz	26624
GeForce GTX 570	1.56 GHz	23040

with subsequent driver updates from NVIDIA, the CUDA performance of newer GPU models such as the GeForce GTX 970 and GeForce GTX 1080 will improve. Nonetheless, this trend presents an interesting conundrum that may be worth further investigation in future research.

Irrespective of the unexpected differences in runtimes between the three GPU models, the maximum runtime for the parallel algorithm in any configuration remains below 0.3 seconds. Even at dataset sizes of 100,000, the runtime of the parallel algorithm continues to grow slowly. For the datasets sampled randomly from the Gaussian and Poisson distributions, the runtime values are extremely low. This is likely due to the reduced number of loop iterations required to cluster these datasets, since they contain a moderate number of duplicate values. The runtimes for the "true random" values display a different trend, and vary based on the linkage metric used. Due to the completely random nature of the values in these datasets, the likelihood of duplicate values or clusters with equal inter-cluster distances is much lower. As such, the number of loop iterations is more likely to reach the worst case value of $N-1$. However, regardless of the distance metric used, the runtimes for the random datasets appear to stabilise by the time the dataset size of 100,000 is reached, and still remain below 300 milliseconds. This demonstrates that, even in the worst-case scenario, the parallel algorithm still offers exceptional performance gains over the classical serial algorithm.

5.5 Implications and Future Work

My proposed algorithm demonstrates the benefits to hierarchical clustering that can be achieved when exploiting the unique characteristics of single-dimensional datasets. By sorting the data and reducing the two-dimensional distance matrix to a single-dimensional distance array, space complexity is reduced from $O(n^2)$ to $O(n)$. Combined with the use of parallel merging, a time complexity of $O(\frac{n^2}{t})$ is achieved, where t is the number of threads that can be run simultaneously. When $t \geq n$, the worst-case runtime is effectively $O(n)$.

When the number of simultaneous threads is sufficient, my proposed algorithm features the same time-complexity and space complexity as the optimal SLINK and CLINK algorithms for single-linkage and complete-linkage. However, unlike the optimal CPU algorithms and other GPGPU algorithms that utilise parallel merging, my algorithm is demonstrated to produce completely equivalent results to the classical algorithm in terms of the Fowlkes-Mallows index, for both single-linkage and complete-linkage. In addition, when clustering datasets with large numbers of identical values or inter-cluster distances, the best-case

runtime of my algorithm is significantly reduced due to the increase in merge parallelism and associated reduction in iteration count.

The approach embodied by my algorithm allows massive single-dimensional datasets to be clustered very quickly on commodity consumer hardware. Unlike traditional distributed computing approaches, no investment in expensive compute nodes is required. The ability to scale to massive datasets allows my algorithm to be utilised in place of existing approaches for handling large datasets, such as BIRCH, CURE, and POP (Dash et al., 2001). Unlike BIRCH and CURE, my algorithm does not sample or summarise any data, and so produces results using the complete amount of available information. My parallel merging approach is similar to the cell-based partitioning approach utilised by CUDA implementation of POP (S. A. Shalom & Dash, 2013), but achieves a higher level of merge parallelism. In addition, my algorithm features reduced space complexity when compared to POP, which utilises two-dimensional distance matrices.

The primary limitation of my proposed approach is that it can currently only be used to cluster single-dimensional data. In order to extend the algorithm to cluster higher-dimensional data, it is necessary to find a method to sort data of arbitrary dimensions that results in an ordering that obeys the properties of a monotonically increasing sequence. Given such a method, the benefits demonstrated by my approach could be applied to facilitate efficient clustering of massive datasets with arbitrary dimensions. I leave this extension to future work.

5.6 Conclusion

Hierarchical clustering is a widely used clustering technique that has been the focus of a large body of existing research. The classical algorithm for the agglomerative hierarchical clustering approach features time and space complexities that make it prohibitively expensive for use in clustering large datasets. Many algorithms have been proposed to improve the efficiency of hierarchical clustering for various linkage metrics. A number of algorithms have also been proposed for scaling hierarchical clustering to large datasets. In recent years, research has focussed on proposing approaches for improving the efficiency of hierarchical clustering through the use of parallelism. The most recent of these approaches utilise GPGPU technologies, which facilitate massive parallelism on commodity consumer hardware.

Very few existing GPGPU implementations perform merges in parallel. The implementations that do perform parallel merging still fall short of maximising the level of merge parallelism. In addition, these approaches do not significantly reduce space complexity over the classical algorithm. Sub-optimal use of parallelism, combined with data transfer overheads for large data structures, limit the performance of these existing GPGPU-based hierarchical clustering implementations.

In this chapter, I proposed a novel GPGPU-based algorithm for hierarchical clustering of single-dimensional data with parallel merging. My proposed algorithm exploits the unique characteristics of one-dimensional data to maximise the level of merge parallelism and significantly reduce memory requirements. I implemented my algorithm using the CUDA language for GPGPU, and described the details of this implementation.

Validation using the Fowlkes-Mallows index of cluster similarity demonstrates that my proposed algorithm produces equivalent results to the classical algorithm for both the single-linkage and complete-linkage metrics. Benchmarking results demonstrate that my algorithm scales well to large datasets, and offers a significant speed-up over a widely-used implementation of the classical algorithm. Future work will look to extend my algorithm and apply the benefits of the proposed approach to data with higher dimensions.

Chapter 6

Input-centric performance model

This chapter builds the foundation for addressing research objective 5. In this chapter, I discuss the factors that contribute to application performance and review existing timing models from the literature on Worst-Case Execution Time (WCET). After examining the limitations of existing models that preclude their use on resource-constrained mobile devices, I propose criteria for determining if a given timing model is amenable to simplification. I evaluate the existing timing models against the proposed criteria and select the count-and-weights timing model, along with elements of the pipeline timing model, for simplification. I then propose an input-centric model of application performance. The proposed model simplifies the selected timing models to reduce their information requirements and computational complexity, making it suitable for use on a resource-constrained mobile device.

After discussing the manner in which the proposed model approximates the information provided by the count-and-weights model, I validate the accuracy of this approximation. Using both synthetic datasets and benchmarking data collected from a range of real devices, I compare the predictors for individual execution paths generated by the count-and-weights model and my proposed model. Validation results demonstrate that my model is an extremely accurate approximation of the count-and-weights model, even as device complexity and OS noise levels increase. The reduced information requirements and low computational cost make the proposed model more adaptable than the existing model, whilst providing a comparable level of accuracy.

This chapter only validates the proposed model for individual execution paths, and does not validate its predictive power for code with multiple execution paths. This more comprehensive validation is subsequently performed in Chapter 7.

The content from this chapter has been submitted for publication as:

- *Rehn, A., Holdsworth, J., Hamilton, J., & Tee, S. An input-centric performance model for mobile applications. Submitted to* Journal of Systems and Software.

6.1 Introduction

Mobile devices and cloud computing are technologies whose prominence continues to grow. The confluence of these technologies represents an area of substantial interest. One common way that mobile devices can utilise the power of cloud computing resources is through the use of computational offloading frameworks (Fernando et al., 2013b). Offloading frameworks automate the process of leveraging cloud compute resources to perform the processing of application tasks. Offloading is performed adaptively, using information about the application and device state to optimise performance and resource usage (Kumar et al., 2013).

As discussed in Chapter 1, existing computational offloading frameworks do not fully account for the influence of an application's input data on its behaviour. The relationship between arbitrary input characteristics and execution behaviour represents a deep level of application-specific knowledge. The utilisation of this knowledge presents an opportunity for improving the decisions made by offloading frameworks. In particular, predictions of application performance may become far more accurate when produced by a model that takes execution behaviour into account (Wilhelm et al., 2008).

Existing models of application performance are often designed for offline use and are poorly suited to the online prediction context of computational offloading. Additionally, many of these models feature information requirements or performance overheads that make their use on resource-constrained mobile devices impractical. To improve the use of application-specific knowledge by computational offloading frameworks, a performance model is needed that takes execution behaviour into account whilst remaining efficient enough for use on mobile devices.

In this chapter, I propose an input-centric model of application performance. The proposed model simplifies the knowledge represented by existing models in order to significantly reduce information requirements and runtime overheads. The model also takes into account the relationship between arbitrary input characteristics and execution behaviour. This combination of efficiency and adaptability provides a foundation for making accurate performance predictions at runtime on resource-constrained mobile devices. These performance prediction can then contribute to high-quality computational offloading decisions.

The contributions of this chapter are as follows:

- I describe the limitations of existing performance prediction models, and identify the characteristics that make a model suitable for adaptation to a mobile device context. Based on these characteristics, I identify which existing models are best suited to such adaptation.
- I present a new model for predicting application performance in an online prediction context. The proposed model draws from existing performance models and simplifies them, in order to provide a cost-effective approximation of the represented information.
- I validate the accuracy of the approximation provided by my model. Validation results demonstrate that the approximation is extremely accurate, both on synthetic datasets and when used with real mobile and embedded devices.

The rest of this chapter is structured as follows. First, I explore the factors that influence application performance and examine the state of existing performance prediction models. Then, I present my proposed model and describe the method by which it simplifies the information represented by existing models. Finally, I validate how well my proposed model approximates the existing model that it simplifies, by comparing the results of the two models on both synthetic datasets and benchmark results from real mobile hardware.

6.2 Background

6.2.1 Application profiling

Computational offloading frameworks perform two key functions. The first is to manage the migration of an application's execution between the local mobile device and a remote server (Shiraz, Sookhak et al., 2015). The second function is to perform cost-benefit analysis to make optimal offloading decisions that satisfy criteria that have been specified by either the developer or the user (Fernando et al., 2013b; Kovachev et al., 2011). In order to predict the cost of performing offloading, frameworks commonly monitor resources on the local device in addition to the behaviour of the application being offloaded. Figure 1.2 in Chapter 1 depicts the common components of the computational offloading cost model.

The component of the offloading cost model that I focus on is application profiling. Application profiling seeks to characterise the performance and resource usage of an application. Profiled characteristics often include execution time, battery usage, and memory footprint (Fernando et al., 2013b). Maximising performance and minimising energy consumption are common offloading goals. An application's memory footprint determines the quantity of data that must be transmitted over the network when performing migration between the device and the remote server (Chun et al., 2011). Minimising data transfer is another common goal when making offloading decisions. It is the performance characteristic of application profiling that I focus on in this study.

Application performance is influenced by numerous factors, including characteristics of the application itself and the operating system and hardware that the application is running on (S. Wang et al., 2002). Characteristics of the application itself can include instruction count, basic blocks and execution paths, input data, and system calls (Reistad & Gifford, 1994; Wilhelm et al., 2008). Factors related to the operating system include context switches and pre-emptive scheduling (De et al., 2007), OS clock ticks and timers, system daemons (Tsafrir et al., 2005), Translation Lookaside Buffer (TLB) mosses, page faults and memory swap-ins, and interrupt handlers (Morari et al., 2011). Factors related to the hardware include CPU and memory frequency, dynamic voltage scaling (Ali et al., 2016), CPU instruction pipeline, cache misses (Stappert & Altenbernd, 2000), branch mis-predictions (Colin & Puaut, 2000), memory bandwidth, and disk and network I/O speeds. I expand the application performance model of S. Wang et al. (2002) to incorporate these factors. Figure 6.1 depicts the expanded application performance model.

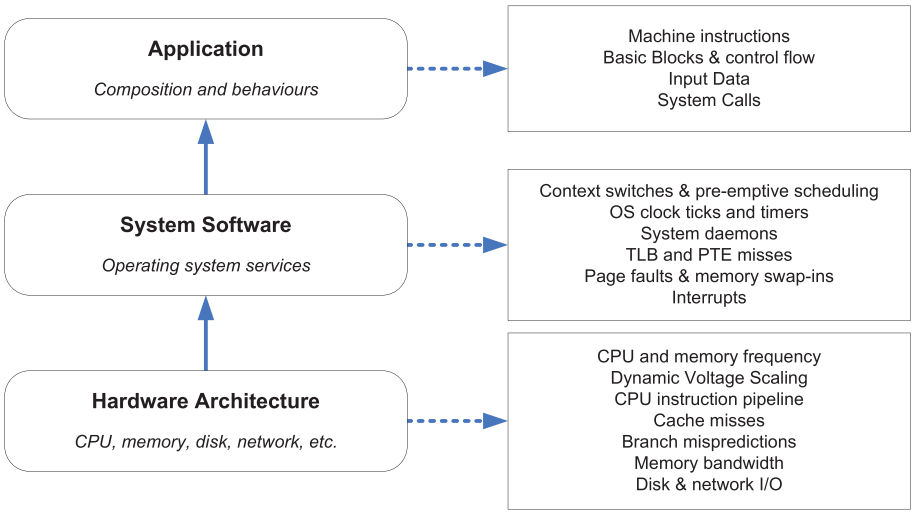


Figure 6.1: Application performance model, expanded from S. Wang et al. (2002)

There are three important observations that arise from the factors in the application performance model when performing application profiling in the context of computational offloading:

- *Performance variability*. Performance anomalies arising from the complexities of modern hardware and operating system architectures are known as OS noise (or OS jitter), and have been studied extensively (Beckman et al., 2008; Hoefler et al., 2010; Morari et al., 2011). Complex interactions between the factors that influence performance can lead to delays that significantly increase execution time when compared to the average or best-case execution time, even when executing code that is *O(1)* in time complexity (De et al., 2007). This variation must be taken into account when predicting execution time based on previously recorded timing data, as is common in a number of computational offloading frameworks (Balan et al., 2003; Cuervo et al., 2010; Flinn et al., 2002).

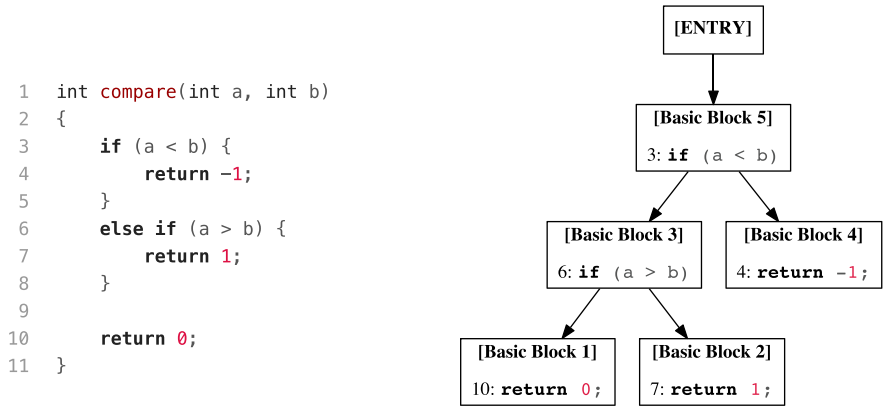
- *Multiple steady-states*. Processors for mobile devices commonly utilise Dynamic Voltage and Frequency Scaling (DVFS) in order to reduce energy usage when the device state satisfies some predetermined condition (Ali et al., 2016; Mittal, 2014). As a consequence, application performance profiles collected when the mobile device is in one steady-state are invalidated when the device moves to a new steady-state. This precludes strategies that utilise pre-computed profiles and necessitates ongoing application profiling over the runtime of the application. As power state changes can be detected programmatically (Pathak et al., 2011), profiling can be automatically triggered at the boundary of each state change.

- *Model complexity*. There are a significant number of factors in the application performance model, a large number of which interact with one another (Stappert & Altenbernd, 2000). Berry, Gracia Perez and Temam (2006) suggest that these interactions may be non-linear. A moderately complete model of application performance is extremely complex, and therefore expensive to utilise for performing predictions at runtime on a resource-constrained mobile device. This suggests the need for a simplified model that trades accuracy for runtime efficiency.

Several of the hardware-related factors are already covered by other components of the computational offloading cost model. The ability to measure certain operating system-related factors may be limited by security restrictions under some mobile operating systems (Levin, 2012). I suggest that a valuable strategy for simplifying the application performance model is to focus exclusively on the factors related to application characteristics. The relationships between these factors can often be determined by offline analysis of an application (Colin & Puaut, 2000), which allows the application profiler to utilise precomputed information to reduce runtime overhead.

6.2.2 Application characteristics

In a compiled application, the smallest unit of execution is a single instruction. When analysing the flow of control through an application, instructions are grouped into units known as basic blocks. Each basic block contains a set of instructions that execute in a linear sequence, without any jumps - that is, they have an entry that represents the first instruction executed, and an exit that represents the last instruction executed (Allen, 1970). Basic blocks are commonly visualised in the form of a Control Flow Graph (CFG), which is a directed graph in which vertices represent basic blocks and edges represent the possible flows of control between basic blocks. A special block is often also added to represent the entry point for the entire CFG (Allen, 1970). CFGs can be used to represent any subset of the code in an application but are most commonly used to represent a single function. Figure 6.2 depicts an example of a CFG for a single function.



(a) (b)

Figure 6.2: (a) Code that compares two integers for sorting purposes, and (b) the corresponding Control Flow Graph (CFG)

Each possible execution of the code represented by a CFG takes the form of a path through the directed graph (Ball & Larus, 1994). CFGs are constructed during the control-flow analysis phase of the compilation process for an application (Aho, Sethi & Ullman, 1986). Due to their ubiquity, CFGs are commonly utilised by software analysis tools (Ernst & Ye, 1997; Kotker, Sadigh & Seshia, 2011; Puschner & Schedl, 1997; Sarkar, 1989; Seshia & Rakhlin, 2008; Theiling, 2002). A CFG does not contain information specifying the conditions required to drive application execution down each possible execution path. This information is not generated automatically by ordinary compilation and must be determined through additional analysis.

Information on the input conditions that trigger specific execution paths can be generated through the use of a static analysis technique known as symbolic execution (King, 1976). In symbolic execution, the inputs of the program are represented as symbols, and the values of all variables at each point during the program's execution are represented as expressions composed of those symbols (Păsăreanu & Visser, 2009). As the program executes symbolically and each branch point is explored, constraints are recorded for each path that represent the input conditions required to reach that path. The set of constraints for a given path is known as its *path constraint*, and is cumulative in nature (King, 1976).

The result of symbolic execution is a data structure known as an *execution tree*. An execution tree represents all of the paths through a program that were explored during symbolic execution and the path constraints for each of those paths. Each node in an execution tree represents an individual statement. Nodes contain both the symbolic values for all variables at that point, as well as the path constraints. Edges between nodes represent transitions between states (King, 1976). An example of an execution tree is shown in Figure 6.3.

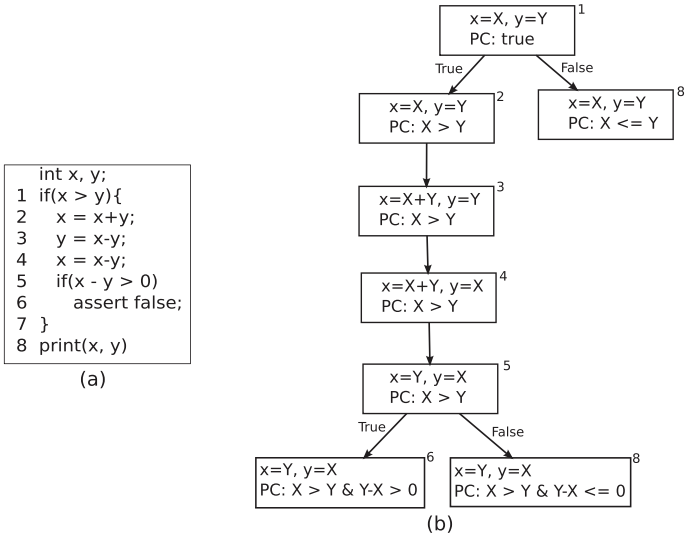


Figure 6.3: (a) Code that swaps two integers, and (b) the corresponding symbolic execution tree, from Anand et al. (2013)

Symbolic execution research is still grappling with a number of open problems (Eler, Endo & Durelli, 2016). One of the biggest limitations is that symbolically executing code that contains loops or recursion can result in path explosion and, in some cases, execution trees that are infinite in size (Păsăreanu & Visser, 2009). Although there have been numerous approaches proposed to deal with this limitation, it is still not a solved problem (Anand et al., 2013). As such, I focus on analysing the behaviour of code that either contains no loops or has had all loops unrolled, and contains no recursive function calls.

Symbolic execution is commonly used for automated generation of test cases for software testing (Cadar & Sen, 2013). Given an execution tree, a constraint solver can be used to generate concrete values to satisfy the path constraints for each path. These concrete values can then be used to drive program execution down the execution path whose path constraints the values were generated to satisfy (Anand et al., 2013). Intuitively, the reverse also holds true. Given a set of concrete input values, the execution tree can be traversed to determine the execution path that those inputs trigger. Accordingly, the information from an execution tree

represents a direct mapping from arbitrary input characteristics to application behaviour. This mapping can be used for behaviour prediction at runtime by utilising a stored representation of an execution tree.

The availability of information generated by symbolic execution represents an opportunity for use in the application profiling components of computational offloading frameworks. However, most existing offloading frameworks that account for input characteristics typically only consider input length, as part of calculating network transfer sizes (Fernando et al., 2013b). I have found only one existing offloading framework that utilises the information from symbolic execution to guide its offloading decisions, which was proposed by C. Wang and Li (2004). However, the timing model utilised by this framework does not adapt to changing device steady-states, and does not take into account the effects of OS noise. To address this gap, I propose an application profiling approach that combines symbolic execution with a more adaptive timing model. My proposed approach utilises execution trees to predict application path behaviours. Once the execution path triggered by a given set of input values has been determined, a timing model that takes into account both changing device steady-states and OS noise can then be utilised to predict the performance of that execution path.

6.2.3 Timing models

Multiple approaches have been explored in existing research for formulating software timing models. Most of these models are drawn from the literature on computing the Worst-Case Execution Time (WCET) of programs, although some models are also focussed on average and best-case execution times. Table 6.1 provides a comparison of a number of these models.

Existing models take a variety of approaches but one attribute is common to a large number of them. Most existing models are designed for offline use, whereby the performance of an application is predicted either before or after it is run on a target hardware architecture. This limits the application of these models to offline performance prediction contexts, which precludes their use in a computational offloading context. In order to adapt these models for use in computational offloading, they require modification to perform prediction whilst an application is currently running. As a result of differing information requirements, existing models vary greatly in their suitability for such modification.

Several existing models cannot adapt automatically to new target architectures without specific additions for each new architecture. The micro-analysis model appears to require hard-coded support for each new architecture (Harmon, Baker & Whalley, 1994). The Branch Target Buffer (BTB) model also requires manual code additions in order to model the branch prediction behaviour of a new architecture (Colin & Puaut, 2000). Although the CFG-based model can adapt to any architecture by reading a machine description file for that architecture, these files must be written by a human for every target architecture, which requires extensive knowledge of the hardware implementation details (Theiling, 2002).

Several models are easily adapted to new architectures but have undesirable features that make them unsuitable for use in a computational offloading context. The implementations of code decomposition model require either user-supplied information on infeasible paths (Park, 1993) or do not provide an automated

Table 6.1
Comparison of existing timing models

References	Model	Required Information	Information supplied by	Limitations
Park (1993); Reistad and Gifford (1994)	Code decomposition	• Source code • Execution time bounds for primitive operations on target architecture	• Compiler • Benchmarks run on the target hardware (Park, 1993) or determined experimentally by paper authors (Reistad & Gifford, 1994)	• Requires benchmarks to be performed on every target architecture, in every steady-state • Requires user-specified information on infeasible program paths (Park, 1993) • Does not take into account the effects of pipelining and OS noise
Altenbernd, Ermedahl, Lisper and Gustafsson (2011); Altenbernd, Gustafsson, Lisper and Stappert (2016); Franke (2008); Giusto, Martin and Harcourt (2001); Holzer, Januzaj, Kugele and Tautschnig (2010); Januzaj, Mauersberger and Biechele (2009)	Count-and-weights	• Instruction stream, either in target architecture instructions or virtual instruction set • Cost weighting for each instruction type	• Compiler • Benchmarks that are run on either the target hardware or a simulator	• Requires extensive benchmarks to be performed on every target architecture, in every steady-state • Does not take into account the effects of pipelining and OS noise
Ernst and Ye (1997); Kotker et al. (2011); Puschner and Schedl (1997); Sarkar (1989); Seshia and Rakhlin (2008)	Graph traversal	• Control Flow Graph • Basic block execution times (Ernst & Ye, 1997; Puschner & Schedl, 1997) or execution path execution times (Seshia & Rakhlin, 2008)	• Compiler • Profiling performed on the target hardware or on a simulator	• Requires benchmarks to be performed over a series of paths through the program, on every target architecture, in every steady-state, or for cycle-accurate simulators to be available for every target architecture
Engblom (2002)	Pipeline	• Instruction stream and Control Flow Graph • Hardware model of CPU pipeline	• Compiler • Generated using profiling on either the target hardware or a simulator	• Requires extensive hardware profiling to be performed either on every target architecture, in every steady-state, or for cycle-accurate simulators to be available for every target architecture
Theiling (2002)	Control Flow Graph (CFG)-based	• Instruction stream • Control Flow Graph • Machine description for target architecture	• Compiler • Reconstructed from the instruction stream • Supplied by the authors	• Requires a machine description for every target architecture that specifies its behaviour, which requires extensive knowledge of every target architecture
Stappert and Altenbernd (2000)	PTA	• Source code • Processor description file	• Compiler • Generated using a simulator	• Requires a cycle-accurate simulator to be available for every target architecture so that processor description files can be generated
Colin and Puaut (2000)	Branch Target Buffer Modelling	• Control Flow Graph • Syntax Tree	• Compiler	• Must be adapted to model the branch prediction details of every target architecture • Only takes into account the effects of branch prediction, not pipelining effects or OS noise
Harmon et al. (1994)	Micro-analysis	• Instruction stream • Machine description for the target architecture that specified hardware behaviour	• Compiler • Created by the authors for the hardware architectures being targeted	• Requires a machine description be created for every target architecture that specifies caching and pipelining effects, which requires extensive knowledge of every target architecture
L. Huang et al. (2010); Y. Kwon et al. (2013)	Key features	• Source code or instruction stream • Execution timings and values for variables and loop counters	• Compiler • Generated using profiling on the target hardware	• Requires extensive profiling to be performed on every target architecture, in every steady-state

means of determining execution time characteristics of the target hardware (Reistad & Gifford, 1994). Despite already performing prediction at runtime, the key features model can only predict the behaviour of larger parts of an application based on the values of the key features. It cannot predict the behaviour of fine-grained components such as individual basic blocks or execution paths (L. Huang et al., 2010).

There are a number of models that are well-suited to being modified for performing predictions at runtime. The count-and-weights, graph traversal, pipeline, and PTA models can all automatically adapt to new target architectures by performing benchmarks on either a cycle-accurate simulator or the real hardware (Altenbernd

et al., 2011, 2016; Engblom, 2002; Ernst & Ye, 1997; Franke, 2008; Giusto et al., 2001; Holzer et al., 2010; Januzaj et al., 2009; Kotker et al., 2011; Puschner & Schedl, 1997; Sarkar, 1989; Seshia & Rakhlin, 2008; Stappert & Altenbernd, 2000). However, because these benchmarks were designed for offline use, they are not optimised for performance. When performing prediction at runtime, it is necessary to perform these benchmarks each time the device changes its steady-state. Accordingly, in order to make the use of these models on resource-constrained mobile devices feasible, it is necessary to modify them so as to minimise runtime overhead. The suitability to this optimisation process varies from model to model.

The PTA and pipeline models both require extensive benchmarking to generate the processor description file or pipeline model, respectively, for a target architecture (Engblom, 2002; Stappert & Altenbernd, 2000). There is no apparent way to simplify the cost of this process for runtime use, based on the details provided. Although this makes these models unsuitable for direct modification, they still provide valuable theoretical constructs. The ability of these models to automatically adapt to new architectures facilitates the integration of individual elements from these models into other, more easily modified models.

Although the graph traversal model is suitable for optimisation in a general sense, each of the specific implementations suffer from characteristics which render them unsuitable. Several older implementations simply assume that operation execution times are known ahead of time (Puschner & Schedl, 1997; Sarkar, 1989), and so feature no benchmarking implementation to modify. The implementation by Ernst and Ye (1997) requires user-supplied information to help drive program execution down all execution paths, which precludes the automation of benchmarking without taking explicit measures to generate the required information. The implementations based on the GAMETIME game theory algorithm (Kotker et al., 2011; Seshia & Rakhlin, 2008) interleave the benchmarking process with the running of the algorithm itself. As a result, benchmarking overhead is intricately linked to the cost of the GAMETIME algorithm.

The count-and-weights model is based on a simple mathematical formula. The formula utilises a set of weights that represent the execution cost of various types of instructions on a given target architecture. The benchmarking, weight-generation, and prediction processes are all separated entirely from one another. The benchmarking and weight-generation processes are expensive and cumbersome, but produce simple information that has the potential to be approximated. As a result of these characteristics, the count-and-weights model provides the most promising avenue for simplification and optimisation.

6.3 Model

As the basis of my own model I utilise the count-and-weights timing model. This model is commonly utilised by approaches that focus on individual instructions when predicting application performance (Altenbernd et al., 2011, 2016; Franke, 2008; Giusto et al., 2001; Holzer et al., 2010; Januzaj et al., 2009). The core mathematical definition of the model, which I refer to as the *count-and-weighting* formula, is depicted in Equation 6.1.

$$t = c1 \times w1 + \ldots + cn \times wn \tag{6.1}$$

Each type of instruction that exists in the compiled program is assigned a weighting, which represents the cost of executing that instruction type on the target architecture being predicted for (Holzer et al., 2010). For each instruction type, the weighting value w_i is multiplied by the c_i, which is the total number of instructions of that type in the compiled code whose timing is being modelled. The summation of these values, t, represents the total predicted time to execute the code being predicted for (Altenbernd et al., 2011; Holzer et al., 2010). Minor variations on this formula have been applied to both instructions and basic blocks (Altenbernd et al., 2011; Ernst & Ye, 1997; Giusto et al., 2001; Holzer et al., 2010; Sarkar, 1989). In one variation, a constant start-up time is added to this formula (Altenbernd et al., 2011). Although all variations of the count-and-weighting formula provide a simple linear model of timing with relation to compiled instructions, the authors of previous works that utilise this formula acknowledge that it does not take into account the effects of hardware architectural features such as CPU pipelining (Altenbernd et al., 2011; Holzer et al., 2010).

Engblom (2002) provides an approach for quantifying the effects of CPU pipelining, which I refer to as the *pipeline timing model*. This model is depicted in Figure 6.4. This model represents timing effects due to pipelining as edge values associated with nodes in a graph, where nodes represent individual instructions. Positive timing values indicate a slowdown due to pipelining effects and negative timing values indicate speed-up. Pipelining effects are directly related to the order of instructions in a stream, and can include both pairwise timing values between instructions that are immediate neighbours and long-running timing values that span the execution of multiple instructions (Engblom, 2002). Because timing values are specific to a given sequence of instructions, this pipeline timing model cannot be directly integrated with the count-and-weighting formula in its original form, since the aggregation of instructions into counts based on instruction type discards sequence order information.

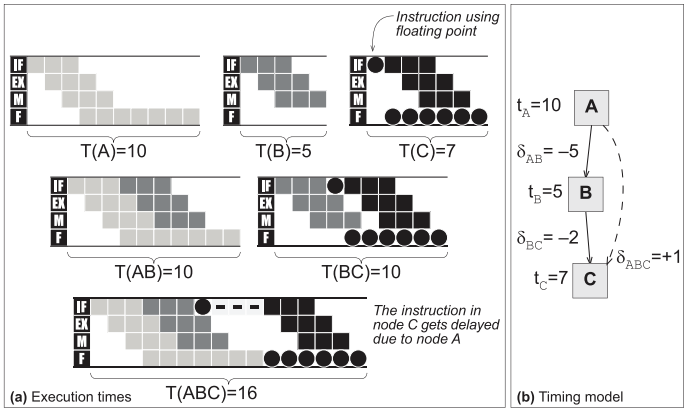


Figure 6.4: Pipeline interference over three nodes and its timing model, from Engblom (2002).

In order to transform the count-and-weighting formula into a form compatible with the pipeline timing model, it must be expanded to include the ordering information that was lost during aggregation. First, let T be the set of unique instruction types, and *cost* be a function that returns the weighting value for a given

instruction type for the desired target architecture:

$$T = \{ type1, type2, \ldots, typen \} \tag{6.2}$$

$$cost : T \to \mathbb{I} \tag{6.3}$$

Next, I define I as the finite sequence of instructions in an execution path, and $type$ be a function that returns the type of a given instruction:

$$I = \{ instruction1, instruction2, \ldots, instructionn \} \tag{6.4}$$

$$typeof : I \to T \tag{6.5}$$

Given the existing set of weighting values w, from the count-and-weighting formula, I then define a finite sequence of weights W where the element at position i represents the weighting value for the type of the instruction I_i:

$$W = \{ cost(typeof(i)) \mid i \in I \} \tag{6.6}$$

With this new sequence, I restate the original count-and-weighting formula as follows:

$$t = \sum W \tag{6.7}$$

The weights in the sequence W can now form the nodes in the pipeline timing model, and the set E of timing effect edges can be added. Evaluating the total execution time for a given path through the graph is straightforward. Simply sum the values of the nodes that form that path, along with their associated edges:

$$t = \sum E + \sum W \tag{6.8}$$

This provides an instruction-based timing model that accounts for the hardware effects of CPU pipelining, but it does not account for the other causes of OS noise, some of which are operating system-related factors. To represent these additional factors, I add a random constant *noise*, which is unique to each invocation of a given execution path:

$$t = noise + \sum E + \sum W \tag{6.9}$$

I name the model represented by Equation 6.9 the *combined timing model*. This model facilitates accounting for both pipelining effects and other contributing factors of OS noise. However, the model still contains a great deal of complexity. The weighting values for each instruction type must be determined for the target architecture that predictions will be generated for, which has previously been accomplished through extensive benchmarking (Holzer et al., 2010). The values for the pipelining effects for each possible sequence of instructions must also be determined, which has previously been accomplished through the use of CPU simulators (Engblom, 2002). The use of either simulation or extensive benchmarking to characterise any given target platform, in all of its possible steady-states, represents a significant obstacle to portability. It is therefore necessary to simplify the combined timing model into a form that suits the criteria identified earlier in my review of the literature. This simplification needs to address the two sets, E and W, for which values are required.

The set of pipeline timing effects E is fixed for a given execution path and a given initial state (Engblom, 2002). When applying the combined timing model to a given execution path, the sum of the set E can therefore be represented by an unknown constant that is unique to that execution path. Since the effects of OS noise are also represented by the random constant *noise*, the definition of *noise* can be expanded to encompass the effects described by the values in E:

$$t = noise + \sum W \tag{6.10}$$

Although techniques exist for measuring OS noise during the execution of an application (De et al., 2007; Morari et al., 2011), these techniques introduce overheads that are too costly when performing measurement on a resource-constrained mobile device. Instead, the variations caused by OS noise can be more cheaply identified through the use of repeated measurements of execution time (Beckman et al., 2006). To report a single value that typifies the execution time for a given execution path, either the mean or median of repeated measurements may be selected (Staelin, 2005). Extreme outliers generated by OS noise may also be removed prior to taking the mean or median using a technique such as the one described in Chapter 4.

Determining the actual values for the set W is time-consuming and expensive. Existing approaches to determining these weightings have included the use of manufacturer's data sheets for CPU models (Harmon et al., 1994), CPU simulation (Altenbernd et al., 2011), and extensive benchmarking (Holzer et al., 2010). The multiple steady-states of modern CPUs that utilise features such as dynamic voltage scaling make the data gathering requirements of these approaches too cumbersome to be practical. As such, it is necessary to approximate these values using the information available. The size of the set W is simply the number of instructions in a given execution path, which is known at compile time. After using repeated measurements to eliminate the random constant *noise*, a single value can be determined for the total execution time t of the execution path. Given the size of the set W and the total execution time t, the simplest approximation of the values in W is to determine their mean. The formula can be modified to become:

$$t = noise + k \times |W| \tag{6.11}$$

Where k is an approximation of the mean of the values in set W:

$$k \approx \frac{\sum W}{|W|} \tag{6.12}$$

This approximation is conceptualised visually in Figure 6.5.

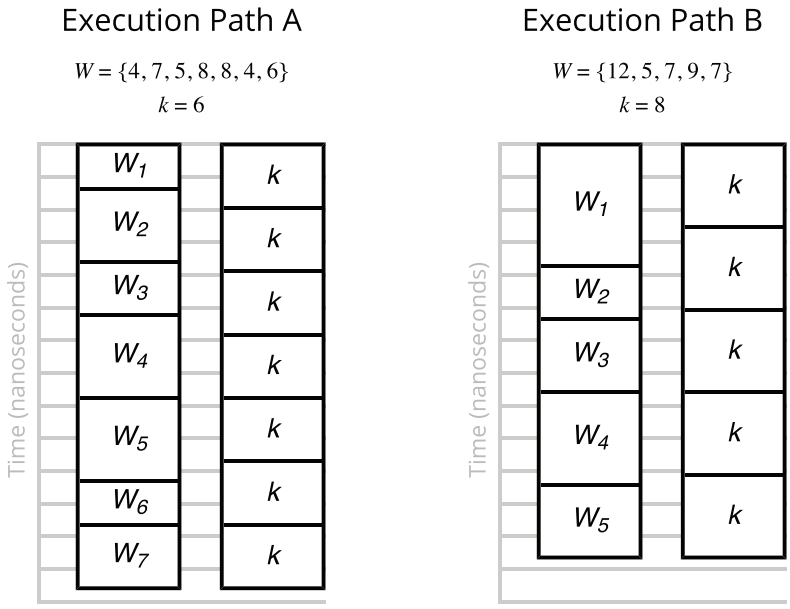


Figure 6.5: Visualisation of the manner in which the k value for a given execution path is an approximation of the mean of the set W for that execution path. The k value for a path is a reflection of the unique mix of instructions in that path.

This simplifies the timing model to represent the execution time of a given execution path as a linear function of its instruction count. The gradient of the linear relationship is specific to each execution path and is a reflection of the unique mix of instruction types and their order within the execution path. Computation of the value k for a given execution path is cheap and requires only the precomputed instruction count and results of repeated measurements of the execution time of that execution path.

Furthermore, I expand my notion of a linear timing model based on instruction count to the level of an entire function. Given the instruction counts and typical execution times of a number of execution paths within a function, it is straightforward to perform least-squares linear regression to fit a single value to the function-wide relationship between instruction count and execution time. This value is conceptually an aggregation of the k values of the execution paths within the function, and logically its accuracy as a predictor of execution time is therefore a direct reflection of the similarity between these k values. For functions whose execution paths feature similar mixes of instruction types and ordering effects, this function-wide approximation should serve well. In cases where execution paths vary significantly in instruction type make-up and ordering effects, accuracy will be diminished. In both cases, performing predictions is extremely cheap and requires only the evaluation of a single linear function. More importantly, predictions can be made for an execution path even when only the k values for other execution paths in the function are known. This provides a simple timing model, suitable for use on resource-constrained mobile devices, that still takes into

account the context-dependence of execution time.

6.4 Validation

To validate how well my proposed model approximates the information provided by the count-and-weights model, I ran both models on two groups of test datasets. The first group contained synthetic datasets, whilst the second group contained datasets collected by benchmarking the performance characteristics of real devices. The composition of the datasets was the same across both groups. Each dataset contained a series of test cases. Each test case consisted of instruction counts for a set of instruction types and an overall runtime. Table 6.2 illustrates the structure of a dataset, with example values.

Table 6.2
Structure of a dataset. Values are examples.

Test Case	`add`	`sub`	`mul`	`div`	`load`	`store`	Runtime
Case 1	43	6	10	45	14	38	862
Case 2	39	6	13	12	43	45	963
Case 3	31	9	9	24	14	43	759
Case 4	37	12	24	13	19	5	469

For each dataset, I generate instruction weights for the count-and-weights model by following the method described by Altenbernd et al. (2016). The test cases are fed into a linear optimisation algorithm to generate individual weights. Both the Least-Squares Fitting (LSQ) and Simulated Annealing (SA) optimisation algorithms have been used previously for this purpose (Altenbernd et al., 2011; Giusto et al., 2001). When run on the test datasets, I discovered that LSQ produced negative weighting values for a number of instruction types in some datasets, whereas SA produced only positive weights. This is consistent with the findings reported by Altenbernd et al. (2016). As such, I utilise only the results of the SA algorithm in my validation.

For each test case in a dataset, I use my proposed model to compute the k value for that test case, given the total instruction count and runtime. I then use the instruction weights generated for the overall dataset to generate set W for that test case. To quantify how well my model approximates the count-and-weights model, I then compute the absolute difference between the k value and the mean of the set W, which I refer to as the *k error*. These validation results are discussed for the two groups of test datasets in the sections that follow.

6.4.1 Synthetic datasets

To represent an ideal device with no OS noise or pipelining effects, I generated a series of synthetic datasets. First, a set of arbitrary instruction types was defined. For each dataset, weights were randomly assigned to each instruction type within the set. Test cases were then generated consisting of random instruction counts. The runtime for each test case was computed using the count-and-weights formula, using the weights for the dataset that contained the test case. Validation was performed on each dataset as described above.

Across all synthetic datasets, the k value for each test case was a perfect approximation of the mean of the set W for that test case. For datasets where the weights generated by SA were identical to those used to construct the dataset, the k error for all test cases was exactly zero. For some datasets, the weights generated by SA deviated from the original weights used to construct the dataset by an extremely small value, typically in the order of 10^{-11}. In these cases, the k error for the test cases in that dataset was a non-zero value, typically in the order of 10^{-14}. These negligible error values demonstrate the mathematical soundness of my model as an approximation of the count-and-weights model, when applied to an idealised context where OS noise and pipelining effects do not exist.

6.4.2 Device datasets

To validate my model in a real-world context, I collected the second group of test datasets by benchmarking the performance characteristics of real hardware. I utilised a series of devices that ranged in complexity from simple embedded microcontrollers to fully-fledged consumer mobile devices. The simplest of these devices was the Arduino Uno R3. Due to memory limitations on this device, it was necessary to write a series of small benchmarks for it by hand. For all other devices, I utilised synthetic mathematical benchmarks that were procedurally generated to provide a random mix of instruction counts for a set of common instruction types. These benchmarks were run on the physical devices to collect their runtimes.

The simplest two devices, the Arduino Uno R3 and the NXP Kinetis FRDM-K20D50M development boards, do not feature robust hardware performance counters. When running benchmarks on these devices, code was added to trigger high and low voltages on the devices' General-Purpose Input/Output (GPIO) pins, to mark the start and end of benchmarks. A digital oscilloscope was connected to the triggered GPIO pins, and the detected changes in voltage were utilised to collect timing information. For all other devices, hardware timing counters provided by the device's CPU were used to collect timing information. The instruction counts and mean runtimes (in nanoseconds) for the benchmarks run on a given device formed the test cases for that device's dataset.

Neither the Arduino Uno R3 or the NXP Kinetis FRDM-K20D50M run full operating systems. The only source of OS noise that can occur on these devices are hardware interrupts. When performing benchmarking on these two devices, I disabled these interrupts, thus eliminating OS noise completely. For all of the remaining devices, I measured the levels of OS noise present during benchmarking using the method described in Chapter 3. The validation results for all of the devices, including OS noise levels and mean k error, are listed in Table 6.3.

Table 6.3
Validation results for real hardware devices

Device	Processor	Clock Speed	Operating System	OS Noise level	Mean k Error
Arduino Uno R3	Atmel ATmega328P	16Mhz	None	0	0.634
NXP Kinetis FRDM-K20D50M	MK20DX128VLH5 (ARM Cortex-M4)	20Mhz	None	0	0.042
Apple iPad 3	Apple A5X (ARM Cortex-A9)	1Ghz	Apple iOS 9.3.4	84	0.014
Apple iPhone 5	Apple A6 (ARMv7-A)	1.3GHz	Apple iOS 9.3.4	1376	0.063
Apple iPhone 6 Plus	Apple A8 (ARMv8-A)	1.4GHz	Apple iOS 9.3.4	5459	0.003
Apple Macbook Pro (Retina, 15-inch, Late 2013)	Intel Core i7-4850HQ	2.3GHz	Apple Mac OS X 10.11.1	9021	0.017

Across all devices, the mean k error was less than one nanosecond. Due to the different treatment required for the Arduino Uno R3, the error level for this device should not be directly compared to the other devices and is included only for completeness. For all remaining devices that used a consistent treatment, the mean k error was less than 0.1 nanoseconds. These results demonstrate that, as hardware complexity and OS noise levels increase, the accuracy of my model as an approximation of the count-and-weights model continues to hold.

6.5 Implications and Future Work

My proposed model demonstrates that it is feasible to adapt existing timing models for use on resource-constrained mobile devices. The discussion of existing frameworks in Section 6.2.3 highlights criteria that determine the suitability of a given timing model for simplification. The method by which I modify the count-and-weights and pipeline models provides an example for how such simplification can be achieved. This strategy, and others like it, could theoretically be applied to other timing models selected using the same criteria.

The validation results presented in Section 6.4 demonstrate that my proposed model is a sound approximation of the count-and-weights model, which requires significantly less information and processing to utilise. Results show that the model remains accurate as hardware complexity and levels of OS noise increase. This demonstrates that a simplified timing model is useful on embedded devices, consumer mobile devices, and on traditional laptop devices. The efficiency and reduced information requirements are a necessity on embedded devices and mobile devices. On traditional laptop devices, these benefits still represent a significant increase in flexibility, due to the ability to quickly adapt to new device steady-states, and a reduction in overheads associated with the extensive data collection required to generate instruction weights. The only data collection that must be performed is to benchmark the performance of the individual execution paths whose performance is being modelled.

The validation results verify the accuracy of my proposed model for individual execution paths, when compared to the count-and-weights model. In Chapter 7, I describe the implementation of a prototype of my model, and perform further analysis to validate the predictive power of the model across entire functions with multiple execution paths. This makes the model general enough for use in computational offloading frameworks.

6.6 Conclusion

Computational offloading is an important technology for improving the performance and battery life of mobile devices by utilising the power of cloud computing resources. Application profiling is a key component of the cost model utilised by computational offloading frameworks to make optimal offloading decisions.

There are a large number of factors that influence application performance and any model that captured them completely would be incredibly complex. Due to modern mobile devices having multiple steady-states, a model is needed that is simple enough to be computed at runtime on a resource-constrained mobile device. The information requirements and processing overheads of existing models make them impractical for use in a mobile context. It is therefore necessary to modify these models in order to simplify them.

There are a number of existing models of application performance that are candidates for simplification. I select the count-and-weights model as a basis, and also incorporate elements of the pipeline model in order to account for the effects of OS noise.

In this chapter, I proposed a model that approximates the predictive power of the count-and-weights model, whilst dramatically simplifying both the amount of information required and the processing overhead, thus making it suitable for use on devices with multiple steady-states.

Validation with synthetic datasets demonstrates that my model is a mathematically sound approximation of the count-and-weights model. Validation with data collected from real devices demonstrates that my model continues to be an accurate approximation of the count-and-weights model even as hardware complexity and OS noise levels increase.

Chapter 7

Input-centric profiling and prediction

This chapter builds on the foundation provided by Chapter 6 to implement a prototype of my proposed input-centric performance model and validate its predictive power for code with multiple execution paths, addressing research objective 5. In this chapter, I propose a language- and platform-agnostic tooling pipeline that can be utilised to implement the proposed model for any programming language or mobile device platform. I then describe the implementation of a prototype of my proposed model, targeting the C++ programming language and the Apple iOS platform.

I validate the predictive power of my proposed input-centric performance model using a series of code modules, running on 20 Apple iPad Air devices. In addition, I define a heuristic to quantify the suitability of a given piece of code to prediction using the proposed model. Validation results demonstrate that the proposed model produces extremely accurate predictions, even as the value of the suitability heuristic becomes less desirable. The accuracy and efficiency of the proposed input-centric performance model make it well-suited for use in a computational offloading framework. The use of rich application-specific knowledge presents opportunities for improved offloading decisions and enhanced application performance.

The content from this chapter is planned to be submitted for publication as:

- *Rehn, A., Holdsworth, J., Hamilton, J., & Tee, S. Input-centric performance prediction for mobile applications. To be submitted to* Journal of Systems and Software.

7.1 Introduction

In Chapter 6, I proposed an input-centric performance prediction model to address the limitations of existing profiling and prediction models. The proposed model utilises automated analysis techniques to fully account for the influence of input data characteristics on an application's behaviour. The proposed model

also simplifies existing timing models to substantially reduce information requirements and performance overheads, making it suitable for use on resource-constrained mobile devices. Validation of the proposed model on both synthetic datasets and real mobile hardware demonstrated that the model is an extremely accurate approximation of the timing models that it simplifies, for individual code execution paths. However, the previous chapter did not validate the predictive power of the proposed model for code with multiple execution paths.

In this chapter, I propose a language- and platform-agnostic tooling pipeline that can be used to implement my input-centric performance model for any programming language and any mobile device platform. Using this tooling pipeline, I describe the design and implementation of a prototype of the model that targets the C++ programming language and the Apple iOS mobile platform. I then validate the predictive power of the model for code with multiple execution paths, running on consumer mobile devices. In addition, I define a heuristic to measure the suitability of a given piece of code for prediction with my model and examine the influence of the suitability value on the accuracy of the generated predictions.

The rest of this chapter is structured as follows. First, I describe the language- and platform-agnostic tooling pipeline, and discuss the implementation of a software prototype using this pipeline. I then describe the experimental methodology used to validate the predictive power of the model for code with multiple execution paths. Finally, I present the results of my validation and discuss the implications of this research.

7.2 Implementation

For predicting the performance of any given piece of code using my model, two key pieces of information are required:

- The execution tree for the code, which is used to determine the execution path that will be triggered by a given set of inputs; and
- The set of instructions for each execution path within the code.

Once collected, this information must be transformed into a machine-readable form that can be compiled into a mobile application, and utilised at runtime once the application is deployed to a mobile device. In this section, I describe the design and implementation of a pipeline of tools to automate the process of collecting and transforming the required data. Although the conceptual design of the pipeline is entirely language-agnostic, I target my implementation at predicting the performance of C++ code. I select C++ for two important reasons. First, C++ is supported natively under the Apple iOS platform, is supported under the Google Android Platform via the Native Development Kit (NDK) (Google Inc., 2016), and is supported under the Microsoft Windows Mobile platform via the the Universal Windows Platform (Microsoft Corporation, 2016). This provides maximum portability amongst popular mobile platforms, giving my implementation the widest applicability available whilst supporting only a single programming language. Second, native languages such as C++ run closer to the underlying hardware than languages that utilise an application-layer

Virtual Machine (VM), such as Java. In the case of Java, the influence of the application-layer VM has been shown to have significant impacts on performing fine-grained benchmarking (J. Y. Gil et al., 2011). The high-level structure of the tooling pipeline is depicted in Figure 7.1.

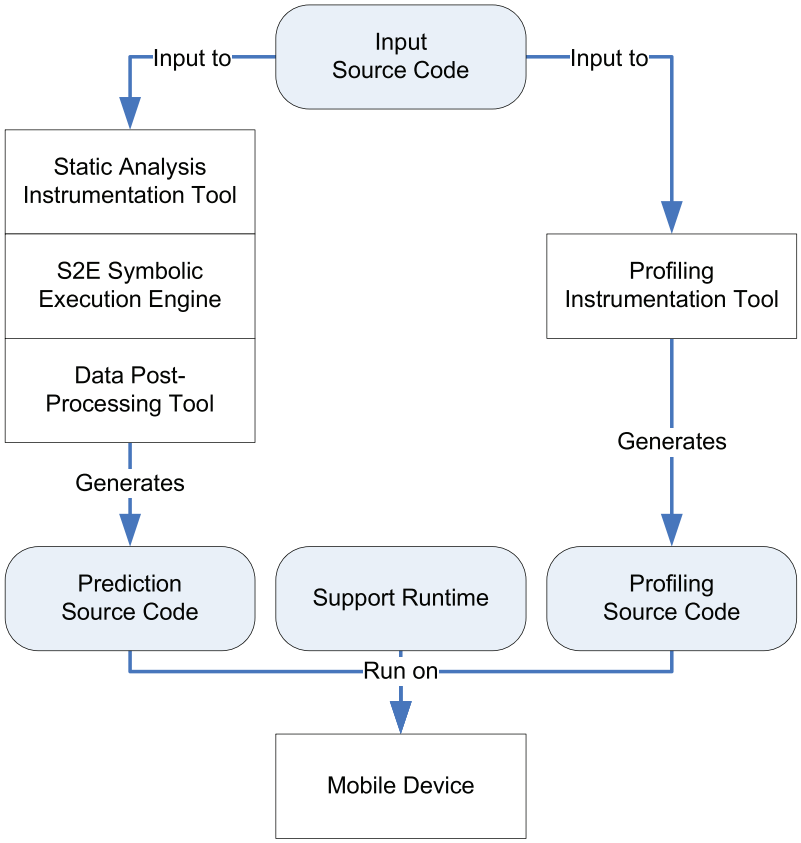


Figure 7.1: High-level overview of the tooling pipeline for collecting and transforming the data required by my input-centric application performance model.

The pipeline is split into two parallel sections. The first section comprises the tools that collect and transform the execution tree. The second section comprises the tools that collect and transform the set of instructions for each execution path through the input source code. The output of of these two pipeline sections is then combined with a code library containing the infrastructure for performing predictions at runtime. The two pipeline sections, and the runtime library, are each described in detail in the sections that follow.

7.2.1 Pipeline section 1: collecting and transforming the execution tree

7.2.1.1 Collecting the execution tree

To generate the execution tree for a given piece of code, I utilise the S2E symbolic execution engine (Chipounov, Kuznetsov & Candea, 2011). S2E is an extension of the earlier KLEE symbolic execution engine (Cadar, Dunbar & Engler, 2008), to which it adds a large number of features. Amongst these features is the ability to analyse the components of an entire software stack, including operating systems and user applications (Chipounov, Kuznetsov & Candea, 2012). The binary translation technique underlying this

feature makes S2E flexible enough to analyse software written in any compiled language. This provides avenues for further extensions of my research, without the need to select another symbolic execution engine. In addition to this, S2E features a number of characteristics that simplify the process of integrating it into my tools pipeline:

- S2E provides a plugin Application Programming Interface (API) that allows me to easily add my own functionality to the engine without the need to modify S2E.
- S2E implements a number of custom opcodes that allow the application being analysed to communicate with the S2E engine (Chipounov et al., 2012). This facilitates runtime coordination between analysed code and engine plugins during analysis.
- S2E inherits from KLEE the ability to automatically generate test cases, which consist of sets of input data that will drive execution down each of the execution paths in the analysed code (Cadar et al., 2008). Access to test case generation is provided through a custom opcode (Chipounov et al., 2011), allowing an engine plugin to collect and store the generated sets of input data.
- S2E represents path constraints using the KQuery grammar of KLEE (Cadar et al., 2008). However, the KLEE API that is exposed to engine plugins provides the ability to access the Abstract Syntax Tree (AST) of each constraint during analysis. This removes the need to parse the KQuery grammar and facilitates the storage of path constraints in a form that is easily consumed by subsequent tools in the pipeline.

The first section of my tools pipeline contains two stand-alone tools and an S2E engine plugin. The first stand-alone tool is the *Static Analysis Instrumentation Tool*. The purpose of this tool is to pre-process the original C++ source code of an application to add S2E custom opcodes. S2E provides a header file that exposes custom opcodes to application code as a series of simple functions that can be called. Simple injection of these function calls into the source code is all that is required to utilise custom opcodes when analysing the compiled code using S2E.

The Static Analysis Instrumentation Tool is built on top of the Clang compiler infrastructure (Lattner, 2008). The Clang libraries provide the full functionality of a compiler front-end, including parsing and semantic analysis. In addition, source rewriting functionality is included to simplify the creation of source-to-source transformation tools, which take source code as their input and produce modified source code as their output.

The Static Analysis Instrumentation Tool parses the supplied C++ source code and constructs the Control Flow Graph (CFG) for each function in the code. The entry point of each basic block in the CFG for a function is instrumented with an S2E custom opcode that delivers the details of the function and basic block number to the S2E engine plugin during analysis. This allows the identified execution paths to be associated with their original source code constructs. The Static Analysis Instrumentation Tool also automatically generates the required S2E initialisation code that performs setup actions prior to the commencement of symbolic execution. The modified source code generated by the Static Analysis Instrumentation Tool can then be compiled and executed symbolically by S2E.

During the analysis of the compiled code by S2E, the engine plugin monitors both execution state changes and custom opcode invocations by the compiled code. From the state change events, the engine plugin collects the execution tree and the ASTs of all path constraints. From the custom opcode invocation events, the engine plugin collects the generated sets of input data, as well as a log of which basic blocks are associated with each execution state. The collected data is stored and then consumed by the second stand-alone tool, the *Data Post-processing Tool*. The Data Post-processing Tool performs a series of transformations on the collected data to make it suitable for use by the runtime library.

7.2.1.2 Transforming the execution tree

Each node in the S2E execution tree represents an application execution state captured during symbolic execution. A node contains a state identifier, a path condition expression, and program counter value. By default, each node is associated only with the low-level state of the compiled application, and not with higher-level source code constructs such as basic blocks. Because state information is only captured at points when application control flow diverges (King, 1976), a given node in the execution tree can correspond to multiple basic blocks from different functions. These blocks may include the basic blocks of any functions that are called, exit blocks that are common to multiple execution paths, as well as the special entry block that is added to each CFG (Allen, 1970).

Using the logs collected by the engine plugin, the Data Post-processing Tool adds to each node in the execution tree the list of functions and basic block numbers associated with that execution state. A valuable side-effect of this process is that it makes it simple to detect infeasible paths in the execution tree. Since infeasible paths are not visited by S2E during execution, an infeasible path is represented by a node in the execution tree that has no outgoing edges and no basic blocks associated with that execution state. The Data Post-processing Tool automatically prunes from the tree any infeasible paths that are detected.

Path constraints in the execution tree are stored in the KQuery constraint language of KLEE. The constraint language is simple and compact, and designed with a focus on accurate bit manipulations (Cadar et al., 2008). The Data Post-processing Tool utilises the constraint ASTs to generate optimised C++ code to implement the constraint language semantics. Each path condition is represented by a C++ function that evaluates the constraint against a set of input values, and determines whether or not the specified values satisfy the path condition. Transforming the path conditions into C++ code allows them to be compiled and then evaluated as efficiently as possible.

Once the execution tree and path constraints have been transformed, source code for a C++ class is generated to represent the execution tree. The class contains the C++ code for all of the path conditions, a representation of the transformed execution tree itself, and functionality to determine the execution path that will be triggered by a given set of input values. When invoked with a given set of input values, the class will traverse the execution tree and evaluate each encountered path condition against the input data. Traversal will take the path whose constraints are satisfied by the input values, and resolve the list of basic blocks in the execution path. The generated C++ class also contains the test cases that were generated by S2E and provides functionality to retrieve the set of generated input values.

7.2.2 Pipeline section 2: collecting and transforming the set of instructions

In order to represent the set of instructions for a piece of code, I utilise the LLVM virtual instruction set (Lattner & Adve, 2004). LLVM is used by a number of existing performance prediction systems (Holzer et al., 2010; Januzaj et al., 2009), as are similar virtual instruction sets (Altenbernd et al., 2011, 2016; Giusto et al., 2001). The benefit of translating source code into a virtual instruction set instead of native machine code for a target architecture is that translation need only be performed once, regardless of the number of hardware platforms being targeted (Altenbernd et al., 2016). This improves the portability of my implementation, and allows it to easily adapt to additional target architectures without the need to add additional code to support those architectures.

The second section of my tools pipeline contains one stand-alone tool, the *Profiling Instrumentation Tool*. Like the Static Analysis Instrumentation Tool, the Profiling Instrumentation Tool is built on top of the Clang compiler infrastructure. The Clang libraries are tightly integrated with the LLVM project and provide functionality for generating LLVM bytecode from parsed source code, as well as manipulating the generated bytecode. The Profiling Instrumentation Tool parses the supplied C++ source code and compiles it into LLVM bytecode. The CFG for each function in the source code is then constructed. For each basic block in the CFG for a given function, the Profiling Instrumentation Tool locates the corresponding basic block in the generated LLVM bytecode. The number of virtual instructions for that basic block is then recorded.

Once the set of virtual instructions for every basic block in every function in the code has been collected, the Profiling Instrumentation Tool generates C++ code to represent the mapping between basic blocks and their associated sets of virtual instructions. The generated code includes functionality to efficiently query this information at runtime. Code is also generated to benchmark the execution time of a given execution path through the original source code. The benchmarking functionality depends on both the code generated by the Data Post-processing Tool and the support code provided by the runtime library.

7.2.3 Runtime library

The final component of the tools pipeline is a runtime code library, which I refer to as the *Support Runtime*. The Support Runtime contains functionality for performing the following tasks:

- Measuring the execution time of code, using the most accurate and efficient implementation available for the given mobile device platform;
- Measuring and reporting the levels of OS noise present on a mobile device;
- Computing the k value for an execution path, from the set of virtual instructions for that path;
- Building the linear model for predicting the execution time of any execution path in a function, based on the individual k values for a set of execution paths in that function; and

- Performing prediction of the execution time of a function for a given set of input values, by traversing the execution tree to determine the execution path triggered by those inputs, and then applying the generated linear prediction model to the virtual instruction count for that execution path.

7.3 Experimental methodology

To validate the predictive power of my model, I test the implementation of my prototype using real mobile devices. For my experiments, I target the Apple iOS platform. I select this platform due to the small, well-defined set of hardware configurations it runs on. In contrast to mobile platforms that run on a variety of devices manufactured by multiple vendors, iOS runs only on fixed hardware models produced by Apple. This improves the reproducibility of my experiment, as it is straightforward to obtain the exact device model used in order to produce consistent results.

I run my experiment on 20 identical Apple iPad Air devices, on loan from Apple. All devices are of the exact same hardware model and run a clean installation of Apple iOS 9.3.1. Each device is placed into Aeroplane Mode, which disables all network connections. Screen brightness and volume for each device is set to maximum. All devices are connected to mains power and report a 100% battery charge level at the beginning of the experiment. No other applications are present in device memory during the run of the experiment.

I validate my model by predicting the execution time for a series of small, mathematically-oriented C++ modules. The code for a number of these modules is based upon source code examples provided in the book "Numerical Recipes in C" (Press, Teukolsky, Vetterling & Flannery, 1996). The modules are as follows:

- COMPUTEBESSELS: computes the Bessel function $J0(x)$ (Abramowitz & Stegun, 1964) for any real x
- EVALUATEPOLYNOMIALS: evaluates polynomial functions
- MULTIPLYVECTORSBYMATRICES: performs vector-matrix multiplication
- PACKBITVECTORS: performs a series of bit-manipulation operations to pack an array of boolean flags into a single bit-field
- PROCESSPIXELS: performs Gaussian noise reduction (Canny, 1986) on an uncompressed RGB image
- WAVELETFILTERS: applies the Daubechies 4-coefficient wavelet filter (Daubechies et al., 1992) to a vector of input data

To simplify static analysis, all loops in the source code for the modules are unrolled and all function calls are inlined. This ensures there is only one CFG for each of the modules, with a single entry point (Kotker et al., 2011). Each of the modules is processed using the tools pipeline described in the previous section. The generated source code is combined with code that implements the behaviour of the experiment, which I refer to as the *Experiment Code*. The combined code is compiled and deployed to the iPads. At this point, the Experiment Code commences the experiment.

The Experiment Code processes each module in turn. First, the automatically generated test cases for a given module are retrieved. The sets of input values from the test cases are used to drive execution of the module down all of its execution paths. The execution time for each execution path is benchmarked multiple times. To compute the k value for a given execution path, it is necessary to simplify the set of collected runtimes to a single value. Several strategies are implemented so that their effectiveness can be compared:

- Use the mean value;
- Use the median value; and
- Remove extreme outliers caused by OS noise, using the outlier removal technique described in Chapter 4, and then use the mean value of the remaining data points.

For each strategy, the single execution time value for each execution path is combined with the set of virtual instructions associated with that path to calculate the k value. Once the k values for all of the execution paths through the module have been collected, validation can commence.

The Experiment Code performs leave-one-out cross validation to assess how well the predictive power of my model generalises to independent data sets (Stone, 1974). Each of the execution paths in a module is validated in turn. To validate a given path, the k values for all other execution paths in the module are used to build the linear prediction equation. The virtual instruction count for the execution path being validated is fed into the equation to generate a predicted execution time. The predicted execution time is then compared to the set of benchmarked execution times for that path to compute an error value. All collected measurements are stored, and the process is repeated for the next execution path in the module. This entire process is repeated for all of the modules, using all three strategies for simplifying multiple runtime measurements into single values.

7.4 Results

I ran the cross-validation experiment five times, with the same set of 20 devices each time. Five runs was the maximum possible in the timeframe during which the devices were available for use. The same experiment conditions applied to all of the runs. The results were averaged across the 20 devices for each run, and then averaged across the five runs. Table 7.1 depicts the averaged results, grouped based on the strategy used to simplify multiple execution time measurements into a single representative value when computing k values.

The small error values demonstrate that, irrespective of the simplification strategy used, my model generated extremely accurate predictions for all code modules being analysed. Of the three simplification strategies, the strategy of using the median execution time to compute k consistently produced the lowest prediction error values. This is due to the robustness of the median against the influence of outliers, and further validates its use in the existing performance benchmarking literature (Staelin, 2005). The fact that the median strategy outperformed the outlier removal strategy also means that the overheads associated with outlier removal can be avoided entirely, further improving the runtime efficiency of the model.

Table 7.1
Cross-validation results for the validation experiment, averaged across all 20 devices and all 5 runs of the experiment.

Module	Strategy: mean		Strategy: median		Strategy: outlier removal	
	k-coherence	Error (%)	k-coherence	Error (%)	k-coherence	Error (%)
computeBessels	4.29 x10^{-04}	2.19	2.65 x10^{-05}	0.85	7.82 x10^{-04}	1.23
evaluatePolynomials	1.08 x10^{-04}	1.35	4.13 x10^{-04}	1.24	1.19 x10^{-04}	1.34
multiplyVectorsByMatrices	1.40 x10^{-03}	2.11	2.96 x10^{-04}	0.75	1.51 x10^{-03}	1.11
packBitvectors	2.06 x10^{-03}	3.03	1.38 x10^{-03}	2.13	1.59 x10^{-03}	2.14
processPixels	1.55 x10^{-04}	0.50	2.84 x10^{-04}	0.50	4.06 x10^{-04}	0.52
waveletFilters	9.55 x10^{-04}	2.31	1.09 x10^{-04}	0.90	1.96 x10^{-03}	1.56

As stated in Chapter 6, I expect that the similarity or dissimilarity between the k values for the execution paths in a given piece of code may influence the accuracy of performance predictions generated for that code. In order to verify if this is the case, it is first necessary to quantify this similarity or dissimilarity. I define the *k-coherence* of a piece of code as the standard deviation of the set of k values for the execution paths through that code. The lower the k-coherence value, the greater the similarity between the k values, and the more "coherent" the code is. The greater the k-coherence value, the greater the dissimilarity between the k values, and the less "coherent" the code.

Irrespective of the k-coherence value of any given code module, prediction errors remain extremely low, all under 5%. To determine if there is any relationship between the k-coherence value for a module and its prediction error value, I plot the errors against the k-coherence values for all three simplification strategies. These plots are depicted in Figure 7.2. The overall trends demonstrated by these plots validate my expectation and suggest that predictions are generally more accurate for code that is more "coherent". However, a larger set of code modules with a far greater range of k-coherence values would be required to perform proper statistical analysis to validate this relationship. As it stands, it is an interesting trend, but of far less importance than the magnitude of the error values themselves, which demonstrate clearly the predictive power of my input-centric performance model.

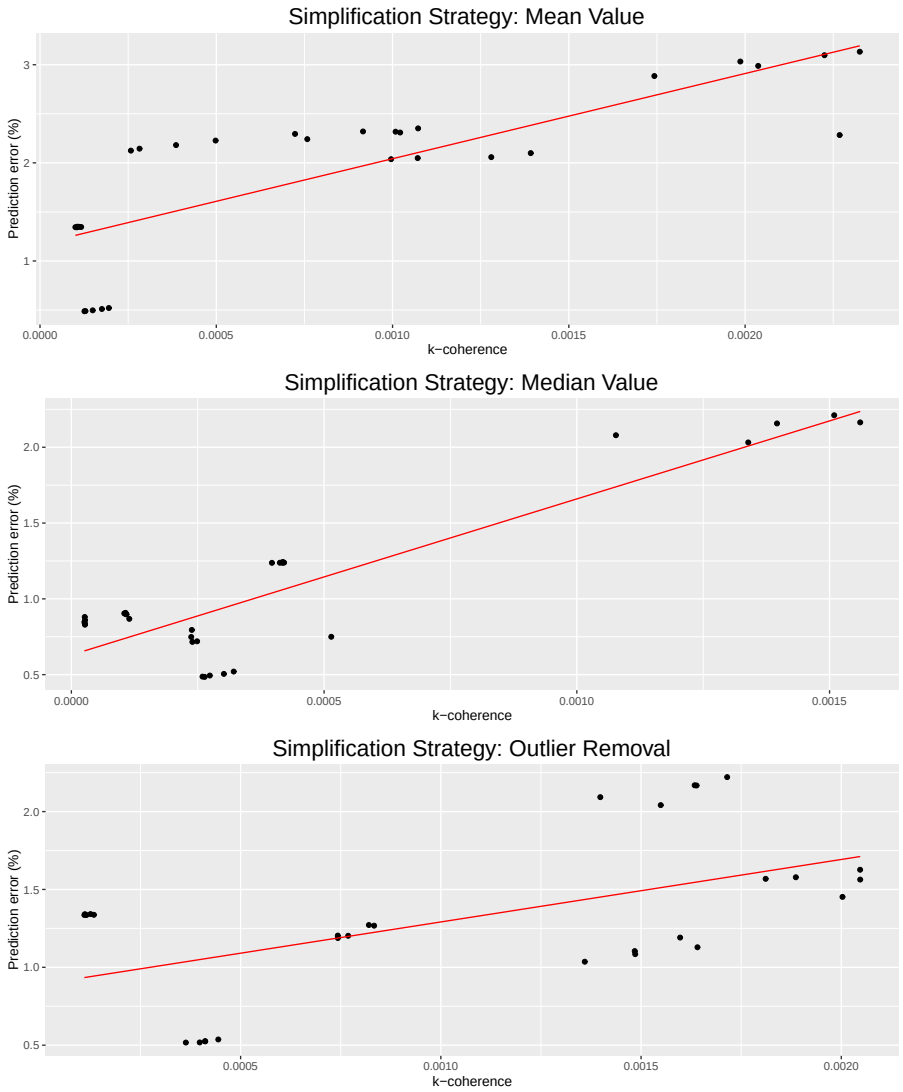


Figure 7.2: Scatterplots depicting prediction error values on the vertical axis and k-coherence values on the horizontal axis, for each simplification strategy. Each point represents the average value for an individual execution path, averaged across the 20 devices and 5 runs of the experiment. Red lines represent the linear function of best fit for each set of values.

7.5 Implications and Future Work

The focus of this study was to implement a prototype of my input-centric timing model, targeting the C++ programming language and the Apple iOS platform. I described a modularised tooling pipeline for implementing this prototype, which identifies and separates each of the key concerns in analysing the source code of an application and generating the information required for performing profiling and prediction at runtime. The design of this pipeline is language- and platform-agnostic and can be utilised to implement my timing model for any programming language or platform, and with any symbolic execution engine. This is facilitated by a number of key design decisions.

The first important design choice that informs the structure of the tooling pipeline is the isolation of the S2E symbolic execution engine. The Static Analysis Instrumentation Tool and the Data Post-processing Tool

isolate the rest of the pipeline from the specifics of the symbolic execution engine being used. This allows alternative implementations of these two tools to be written for any arbitrary symbolic execution engine, providing a drop-in replacement without effecting the rest of the tooling pipeline. In addition to facilitating the use of different symbolic execution engines, different programming languages can be supported by automating the transformations necessary to make those languages compatible with the chosen symbolic execution engine.

The second key design decision is the isolation of platform-specific code to the Support Runtime library. This allows support for new platforms to be added to the Support Runtime, without the need to modify any of the other components of the pipeline. If bindings are created to allow access to the Support Runtime from additional languages, the existing platform support can also be reused when adding support for new programming languages.

In validating the effectiveness of my timing model for code with multiple execution paths, I defined the heuristic of *k-coherence*, which quantifies the similarity of the unique mix of instructions in all execution paths of a function. This heuristic is effectively a measure of instruction heterogeneity across the execution paths of a function. Validation results demonstrate that predictions generated by my timing model are extremely accurate, and remain accurate even as k-coherence values become less desirable.

The combined efficiency and accuracy of my input-centric timing model make it well-suited for use as part of a computational offloading framework. The deep level of application-specific knowledge provided by my approach facilitates adaptation to both arbitrary input data characteristics and changes in device steady-state. Very few of the existing computational offloading frameworks discussed in Section 1.1.2 account for arbitrary input characteristics, and none account for the combination of both arbitrary input characteristics and device steady-state changes. The adaptation provided by the proposed timing model could potentially contribute to improved offloading decisions, resulting in better optimisation of application performance and resource use.

The results presented by this study are extremely promising, but limited to the C++ programming language and the iPad Air device model used for testing. In addition, the range of k-coherence values represented by the tested code is relatively moderate and does not contain any extreme cases. Future work could include testing on a wider variety of devices and mobile platforms and with code that represents a more extreme range of k-coherence values. Application of the proposed tooling pipeline to implementations for additional programming languages would also represent a further extension to this work.

7.6 Conclusion

Application performance profiling is a core component of computational offloading frameworks, which are a prominent technology in optimising the use of cloud compute resources by mobile devices. Existing frameworks often fail to take into account the effect that arbitrary input characteristics have on application behaviour. Although a number of sophisticated performance prediction models are available, most of these are designed for offline use and have limitations that preclude their use in a computational offloading context.

In the previous chapter, I proposed an input-centric performance prediction model that simplifies existing models to provide predictions with significantly reduced information requirements. Previous validation of my model demonstrated that it is an extremely accurate approximation of the models that it simplifies. However, the previous chapter did not validate the predictive power of my model.

In this chapter, I detailed the design and implementation of a working prototype of my input-centric performance model. I then validated the predictive power of the model by testing it on real mobile devices. Validation results demonstrate that my model produces extremely accurate predictions. I also quantified the characteristic of code that determines how well-suited it is for use with my prediction system. Results demonstrated that the prediction error remains extremely low, even as the suitability metric value becomes less desirable. The accuracy and efficiency of my model represents a significant improvement to the information available to computational offloading frameworks, and has the potential to result in higher-quality offloading decisions, leading to improved application performance and resource usage.

Chapter 8

Conclusion

8.1 Thesis Outcomes

8.1.1 Chapter 2

Chapter 2 describes the DSRM used by the research in this thesis. The focus of this discussion is the methodology used to evaluate research artifacts that target mobile device platforms. Evaluation of design science artifacts is guided by the evaluation methodologies of existing research disciplines. Application of these evaluation methodologies is tailored to the characteristics of the artifact being evaluated. There are a number of unique technical challenges involved in the evaluation of artifacts for mobile devices that are not addressed by the existing design science literature. These challenges are described in detail in Section 2.2.3.

In this chapter, I propose an automated data collection and processing framework. The design of this framework is informed by requirements drawn from the existing literature on both artifact evaluation methodology and mobile data collection frameworks. The proposed framework supports both formative and summative artifact evaluation in naturalistic and artificial environments. The framework utilises a client-server model to provide centralised storage and processing of collected data. An artifact deployment component simplifies the process of deploying and managing artifact evaluation instances on devices belonging to both researchers and study participants. Through the combination of these features, the proposed framework facilitates an automated, extensible, and customisable workflow for evaluating artifacts that target mobile device platforms.

The evaluation workflow enabled by the proposed framework automates the feedback loops of the DSRM. This automation facilitates more rapid iterations and provides richer information to researchers. These benefits allow for the development of more mature, refined artifacts. The extensible and customisable workflow can be tailored to suit the needs of all stakeholders, simplifying the process of complying with organisational policies.

The validation of the automated data collection and processing framework is embedded in this thesis. The prototype implementation of the proposed framework was utilised in the data collection for subsequent

chapters of the thesis. Without it, the collection and processing of data from such a large number and variety of mobile devices simply would not have been feasible.

8.1.2 Chapter 3

Chapter 3 addresses research objectives 1, 2, and 3, and addresses part of research objective 4. The two outcomes of this chapter are an examination of the characteristics of OS noise on mobile devices and the development of a technique to mitigate the effects of OS noise on micro-benchmarking results.

In the first part of this chapter, I conduct a study of OS noise on Apple iPad Air devices. The study includes an experiment that measures the levels of OS noise present on a device, whilst systematically altering the device's steady-state. Examination of the noise profiles collected from the devices demonstrates that the observed characteristics of OS noise on mobile platforms are consistent with the characteristics observed on traditional desktop and server platforms in the existing literature. This addresses research objectives 1 and 2. Analysis of the collected data also demonstrates that a relationship does indeed exist between levels of OS noise and changes in device steady-state. This addresses research objective 3.

In the second part of this chapter, I propose an adaptive noise mitigation technique for use with micro-benchmarking data on mobile platforms. The proposed technique measures the levels of OS noise present on a device and adaptively adjusts micro-benchmarking granularity to achieve the greatest possible level of accuracy, whilst mitigating the effects of OS noise on the collected data. This allows meaningful results to be obtained from micro-benchmarking on mobile devices by adapting to changes in OS noise each time the device changes steady-state. This addresses all of the requirements of research objective 4, except for the requirement to be fully automated. The outlier removal step described in this chapter is only partially automated and is replaced with a fully automated outlier removal technique in Chapter 4.

8.1.3 Chapter 4

Chapter 4 builds on the adaptive noise mitigation technique presented in Chapter 3. In this chapter, I propose a fully automated outlier removal technique to replace the partially automated approach described in the previous chapter. Before developing the outlier removal technique itself, I first collect OS noise profiles from a variety of Apple iOS device models. This additional data collection supplements the OS noise study performed in Chapter 3, further contributing to research objectives 1 and 2. After examining the collected data, I describe the nature of the outliers that are present and discuss the nested clustering structure that can be observed in the data. To capture this nested structure, I select hierarchical clustering as the basis for my proposed outlier removal technique. I then propose a heuristic algorithm that exploits the nature of the outliers present in a dataset to automatically determine where to cut the dendrogram produced by hierarchical clustering, and remove outliers based on this cut.

In the later part of this chapter, I propose a simplified heuristic that utilises precomputed data to substantially reduce the computational complexity of the proposed outlier removal technique. Validation of the two

heuristics demonstrates that the simplified heuristic provides a very accurate approximation of the full heuristic. The accuracy and efficiency of the simplified heuristic makes it feasible to perform fully-automated outlier removal on resource-constrained mobile devices. This addresses the final requirement of research objective 4.

8.1.4 Chapter 5

Although the automated outlier removal technique proposed in Chapter 4 is extremely efficient, the hierarchical clustering technique that it relies on is too computationally expensive for use on resource-constrained mobile devices. Chapter 5 addresses this limitation by proposing a novel GPU-accelerated parallel algorithm for hierarchical clustering of single-dimensional data. Existing parallel implementations of hierarchical clustering do not maximise the benefits achieved through parallel processing because the parallel portions of the algorithm are bottlenecked by the steps that are performed in sequence. In particular, existing parallel implementations typically perform the merging of clusters in serial during part or even all of the clustering process. My proposed algorithm exploits the unique characteristics of single-dimensional data to maximise the number of merges that can be performed in parallel. This minimises the portions of processing that are performed in serial, eliminating bottlenecks and allowing the algorithm to achieve the full speed-up offered by parallel processing.

I validate both the efficiency and correctness of my proposed algorithm against the classical serial algorithm for hierarchical clustering. Validation results demonstrate that the clusterings produced by my proposed algorithm are equivalent to the results produced by the classical algorithm, for both the single-linkage and complete-linkage metrics. Performance benchmarking also demonstrates that my parallel algorithm achieves a significant speed-up when compared to the classical serial implementation. The maximisation of merge parallelism allows the proposed algorithm to scale extremely well to large datasets. The use of General-Purpose Graphics Processing Unit (GPGPU) technologies in the implementation of my proposed algorithm facilitates massive parallelism on commodity consumer hardware, including modern mobile devices. The efficiency of the proposed algorithm makes the entire automated outlier removal technique, proposed in 4, efficient enough for use on resource-constrained mobile devices.

8.1.5 Chapter 6

Chapter 6 builds the foundation for addressing research objective 5. In this chapter, I explore the factors that contribute to application performance and review software timing models from the existing literature on Worst-Case Execution Time (WCET). Most existing timing models are not designed for use on mobile devices and do not take into account the effects of changing device steady-states. In addition, most existing models feature information requirements and computational overheads that make their use on resource-constrained mobile devices impractical. As part of my review of existing timing models, I discuss the suitability of each model to adaptation for mobile devices. At the conclusion of my review, I select the

count-and-weights timing model as most suitable for simplification. Combined with elements of the pipeline model, I use this model as the basis of proposing a new, simplified timing model for use on mobile devices.

I simplify the count-and-weights timing model to produce an approximation that features substantially reduced information requirements and computational complexity. I also incorporate elements of the pipeline timing model to allow the count-and-weights model to account for changing device steady-states. Validation of the proposed timing model, using both synthetic datasets and benchmarking data collected from a variety of embedded and mobile devices, demonstrates that it is an extremely accurate approximation of the count-and-weights model for individual code execution paths. The accuracy of this approximation continues to hold even as hardware complexity and OS noise levels increase. The reduced information requirements and low computational cost make the proposed model suitable for use on resource-constrained mobile devices, whilst providing a level of accuracy comparable to the unmodified count-and-weights model.

8.1.6 Chapter 7

Chapter 7 builds on the foundation provided by Chapter 6 to implement a prototype of my proposed input-centric performance model and validate its predictive power for code with multiple execution paths, thus addressing research objective 5. In this chapter, I propose a language- and platform-agnostic tooling pipeline that can be used to implement my proposed input-centric application performance model for any programming language and any mobile device platform. Using the proposed tooling pipeline, I describe the implementation of a prototype of my input-centric performance model that targets the C++ programming language and the Apple iOS mobile platform.

I validate the predictive power of my proposed input-centric performance model by performing an experiment on 20 Apple iPad Air devices. The experiment performs profiling and prediction of a series of code modules with multiple execution paths. For each execution path through each code module, the experiment performs leave-one-out cross validation to demonstrate the ability of the proposed model to generalise to independent datasets. In addition, I define a heuristic to quantify the suitability of a given piece of code to prediction using the proposed model. Validation results demonstrate that my proposed input-centric performance model produces extremely accurate predictions, even as the value of the suitability heuristic becomes less desirable. The accuracy and efficiency of the proposed model make it well-suited for use in a computational offloading framework. This satisfies the requirements of both research objective 5, as well as my over-arching research question.

8.2 Theoretical Contributions

This thesis provides a number of theoretical contributions to the greater body of knowledge, particularly with respect to OS noise and application timing models.

The automated outlier removal technique proposed in Chapter 4 identifies the structure of outliers introduced by OS noise and demonstrates that automated removal of these outliers is feasible. The nested clustering structure of noisy micro-benchmarking data is captured using hierarchical clustering, and a heuristic algorithm is utilised that exploits the characteristics of OS noise to automatically determine the point at which the resulting dendrogram is cut. The simplified version of this heuristic demonstrates that the information encapsulated by the proposed approach can be captured and precomputed, facilitating use on resource-constrained mobile devices by reducing the computational complexity of the heuristic to $O(n \log n)$.

Outlier removal utilises a novel GPU-accelerated clustering algorithm, proposed in Chapter 5. The proposed algorithm demonstrates that hierarchical clustering can be performed with significantly lower computational complexity and memory use, if the dataset can be sorted such that the ordered data satisfies the requirements of a monotonically increasing sequence. The properties of a monotonically increasing sequence are also exploited to maximise the extent to which merges can be performed in parallel, thus maximising the speed-up achieved by parallel processing.

The input-centric application timing model proposed in Chapter 6 simplifies the count-and-weights model, and elements of the pipeline model, to substantially reduce information requirements and computational complexity. The information requirements of these models previously made their use on resource-constrained mobile devices impractical. My simplified application timing model addresses this limitation. Additionally, the incorporation of elements of the pipeline timing model allow the count-and-weights model to account for the effects of OS noise.

The language- and platform-agnostic tooling pipeline proposed in Chapter 7 allows my proposed input-centric profiling and prediction system to be implemented for any programming language, and for any target mobile device platform. The modular nature of the pipeline allows any symbolic execution engine to be utilised and simplifies the process of adding support for additional symbolic execution engines to an existing implementation. This flexibility improves the generality of the input-centric profiling and prediction system and allows new advances in symbolic execution techniques to be incorporated very quickly as they become available.

8.3 Empirical Contributions

Analysis of the data collected in the OS noise study described in Chapter 3, and in the evaluation sections of the remaining data chapters, provides a number of empirical contributions to the greater body of knowledge.

The OS noise study in Chapter 3 is the first detailed exploration that I am aware of that examines OS noise on consumer mobile devices. Examination of the collected noise profiles demonstrates that characteristics of OS noise on mobile devices are consistent with those characteristics observed on traditional desktop and server platforms in the existing literature. In addition, analysis demonstrates that a relationship exists between changes in OS noise levels and changes in device steady-state. These insights expand the greater understanding of OS noise to a new class of devices and highlight the key challenges in addressing OS noise on mobile platforms.

Evaluation of the adaptive OS noise mitigation technique proposed in 3 demonstrates that the proposed technique maximises the level of micro-benchmarking accuracy, whilst mitigating the effects of OS noise on the collected data. This facilitates an adaptive approach to micro-benchmarking granularity, guided by the levels of OS noise present on a system at the time that benchmarking is performed.

Evaluation of the novel GPU-accelerated clustering algorithm proposed in Chapter 5 demonstrates an enormous speed increase over the classical algorithm for hierarchical clustering. The optimal level of merge parallelism achieved by the proposed approach allows the computational complexity of the algorithm to scale extremely well for datasets with duplicate values and duplicate inter-cluster distances. This is demonstrated by the benchmarked runtimes of the proposed algorithms, which are observed to be far better for Gaussian- and Poisson-distribution datasets than the algorithm's theoretical worst-case time complexity of $O(\frac{n^2}{t})$, where t is the number of threads that can be run simultaneously.

Evaluation of the input-centric timing model proposed in Chapter 6 demonstrates that the proposed simplification represents an extremely accurate approximation of the information provided by the count-and-weights model. Additionally, the accuracy of this approximation continues to hold across a wide range of devices with increasing levels of hardware complexity and OS noise, demonstrating that the simplified timing model is useful on embedded systems, mobile devices, and even laptops. This accuracy is achieved whilst maintaining substantially reduced information requirements and computational cost when compared to the unmodified count-and-weights model.

Evaluation of the input-centric timing model for code with multiple execution paths in Chapter 7 demonstrates that the predictive power of the model is excellent. Predictions generated by the model remain extremely accurate, even as the dissimilarity between execution paths in a function increases. The results of the leave-one-out cross validation experiment demonstrate that the model generalises well to independent datasets, and the runtime of a given execution path can still be accurately predicted when the model is built using only information about other execution paths in the function.

8.4 Real-world Contributions

The research in this thesis follows the DSRM. The automated data collection and processing framework proposed in Chapter 2 represents the methodological contribution of the thesis. The proposed framework automates the feedback loops of the DSRM for the evaluation of artifacts that target mobile device platforms. The incorporation of insights from the literature on mobile data collection highlights the potential benefits that can be gained through the exaptation of existing practices for use in DSRM evaluation methodology. The automated data collection and processing framework provides a number of benefits to researchers and other stakeholders:

- The automation of the DSRM feedback loops facilitates more rapid iterations. Many more iterations can be performed in a given span of time, which can result in the development of more refined and mature artifacts.
- The central data storage provided by the server component of the framework allows data collected from disparate sources to be aggregated into a single repository. Support for exporting the collected data in standardised formats makes it easier for researchers to comply with organisational data storage policies.
- The modular design of the data processing components allows researchers to incorporate existing analysis tools and software into the automated framework. The re-use of existing tools maximises the intellectual and financial return on investment for those tools.
- Data processing can be tailored to meet the needs of all stakeholders. The modular design of the data processing components allows additional processes, such as report generation, to be incorporated into the automated workflow.

In addition to the methodological contribution, the proposed input-centric profiling and prediction system represents a number of contributions with real-world applications. The approach by which the count-and-weights model is simplified in Chapter 6 demonstrates that it is feasible to adapt existing timing models for use on resource-constrained mobile devices, even when those timing models were not designed for a mobile context and feature limitations that make such use impractical without modification. The discussion of existing timing models in Chapter 6 also provides an example of criteria that can be used to determine the suitability of a given timing model to such adaptation.

The culmination of the research in this thesis is the proposed input-centric profiling and prediction system. The proposed system is extremely efficient. This allows it to invoke processing each time a mobile device changes steady-state, thus adapting to changing device conditions. This adaptation allows the effects of OS noise to be mitigated dynamically, so that accuracy is maximised based on the current device conditions. The use of deep application-specific knowledge allows the behaviour and performance of an application to be predicted with great accuracy. These predictions allow the system to provide high-quality information to a computational offloading framework, which can then use that information to make optimal offloading

decisions, thus maximising the performance and energy efficiency of mobile applications, and improving the user experience.

8.5 Limitations and Recommendations for Future Research

The outcomes of this thesis represent the first exploration of new insights and techniques, incorporating studies and approaches that break new ground. Accordingly, there are a number of limitations that can be addressed by future extensions to this research.

The automated data collection and processing framework presented in Chapter 2 has only been validated as part of the research in this thesis. Future research can extend this validation to incorporate case studies involving other researchers using design science and evaluating artifacts for mobile device platforms. These case studies could also involve evaluation methodologies that incorporate human participants, to test the effectiveness of framework-facilitated interactions between the researchers and the users of mobile devices, which was outside the scope of this thesis.

The OS noise study conducted in Chapter 3 was performed on a small set of devices from a single device family. Future research can extend this study to a wider variety of device models and platforms, to determine if both the observed OS noise characteristics, and the relationship between OS noise levels and device steady-state, are consistent across a broader variety of mobile devices.

The novel GPU-accelerated clustering algorithm presented in Chapter 5 demonstrates the impressive gains in efficiency that can be achieved when clustering ordered data that satisfies the requirements of a monotonically increasing sequence. The current implementation only clusters single-dimensional data, since univariate datasets naturally form a monotonically increasing sequence when they are sorted in ascending order. Future research can extend this approach to datasets with higher dimensionalities by exploring techniques for transforming and sorting higher-dimensional datasets in such a manner that the produced ordering satisfies the requirements of a monotonically increasing sequence. Such an extension would greatly increase the applicability of the proposed clustering algorithm, allowing the benefits of optimal merge parallelism to be applied to a far wider range of datasets.

The input-centric profiling and prediction system described in Chapters 6 and 7 represents the culmination of the research in this thesis. The proposed system was only implemented for a single programming language and mobile platform during the course of this research. However, the tooling pipeline described in Chapter 7 makes it easy to implement the proposed system for any programming language or target mobile platform. In addition, validation of the predictive power of the system was performed using only a small range of code modules. Future research can extend this thesis by implementing my proposed input-centric profiling and prediction system for a wide variety of programming languages and mobile device platforms, and validating its predictive power for a wider range of source code, including real-world mobile applications.

References

Aboutabl, A. E. & Elsayed, M. N. (2011). A novel parallel algorithm for clustering documents based on the hierarchical agglomerative approach. *International Journal of Computer Science and Information Technology (IJCSIT)*, *3*(2), 152–163.

Abramowitz, M. & Stegun, I. A. (1964). *Handbook of mathematical functions: with formulas, graphs, and mathematical tables* (Vol. 55). Courier Corporation.

Agarwal, S., Mahajan, R., Zheng, A. & Bahl, V. (2010). Diagnosing mobile applications in the wild. In *Proceedings of the 9th ACM SIGCOMM workshop on hot topics in networks* (pp. 22:1–22:6). New York, NY, USA: ACM.

Ahnn, J. H. & Potkonjak, M. (2013). mhealthmon: Toward energy-efficient and distributed mobile health monitoring using parallel offloading. *Journal of Medical Systems*, *37*(5), 9957. Retrieved from `http://dx.doi.org/10.1007/s10916-013-9957-0` doi: 10.1007/s10916-013-9957-0

Aho, A. V., Sethi, R. & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Akkan, H., Lang, M. & Liebrock, L. M. (2012). Stepping towards noiseless linux environment. In *Proceedings of the 2nd international workshop on runtime and operating systems for supercomputers* (pp. 7:1–7:7). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2318916.2318925` doi: 10.1145/2318916.2318925

Ali, F. A., Simoens, P., Verbelen, T., Demeester, P. & Dhoedt, B. (2016). Mobile device power models for energy efficient dynamic offloading at runtime. *Journal of Systems and Software*, *113*, 173 - 187. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0164121215002666` doi: http://dx.doi.org/10.1016/j.jss.2015.11.042

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Notices*, *5*(7), 1–19. Retrieved from `http://doi.acm.org/10.1145/390013.808479` doi: 10.1145/390013.808479

Almeida, J., Barbosa, L., Pais, A. & Formosinho, S. (2007). Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering. *Chemometrics and Intelligent Laboratory Systems*, *87*(2), 208 - 217. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0169743907000068` doi: http://dx.doi.org/10.1016/j.chemolab.2007.01.005

Altenbernd, P., Ermedahl, A., Lisper, B. & Gustafsson, J. (2011). Automatic generation of timing models for timing analysis of high-level code. In S. Faucou (Ed.), *Proceedings of the 19th international conference on real-time and network systems (RTNS2011).* The IRCCyN lab.

Altenbernd, P., Gustafsson, J., Lisper, B. & Stappert, F. (2016). Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 1–30. Retrieved from http://dx.doi.org/10.1007/s11241-016-9250-7 doi: 10.1007/s11241-016-9250-7

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... Zhu, H. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86*(8), 1978 - 2001. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121213000563 doi: http://dx.doi.org/10.1016/j.jss.2013.02.061

AnandTech, Inc. (2013). *They're (almost) all dirty: The state of cheating in android benchmarks.* Retrieved 2014-07-23, from http://www.anandtech.com/show/7384/state-of-cheating-in-android-benchmarks

Androulakis, S., Buckle, A. M., Atkinson, I., Groenewegen, D., Nicholas, N., Treloar, A. & Beitz, A. (2009). Archer–e-research tools for research data management. *International Journal of Digital Curation*, *4*(1), 22–33. doi: 10.2218/ijdc.v4i1.75

Angin, P. & Bhargava, B. (2013). An agent-based optimization framework for mobile-cloud computing. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, *4*(2), 1-17. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84886930978&partnerID=40&md5=84bd3c6cd20cbcc7ab44b47a5a22366a

Apple Inc. (2005). *Technical Q&A QA1398: Mach absolute time units.* Retrieved 2014-07-21, from https://developer.apple.com/library/mac/qa/qa1398/_index.html

Apple Inc. (2013). *Memory usage performance guidelines: Tips for allocating memory.* Retrieved 2014-07-08, from https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html

Apple Inc. (2014). *Xcode - apple developer.* Retrieved 2014-03-25, from https://developer.apple.com/xcode/

Apple, Inc. (2016). *Record-breaking holiday season for the app store.* [Press release]. Retrieved 2016-11-14, from http://www.apple.com/pr/library/2016/01/06Record-Breaking-Holiday-Season-for-the-App-Store.html

Arumugavelu, S. & Ranganathan, N. (1996). SIMD algorithms for single link and complete link pattern clustering. In *Pattern recognition, 1996., proceedings of the 13th international conference on* (Vol. 4, p. 625-629 vol.4). doi: 10.1109/ICPR.1996.547640

Aucinas, A., Crowcroft, J. & Hui, P. (2012). Energy efficient mobile M2M communications. *Proceedings of ExtremeCom*, *12*.

Baker, F. B. (1974). Stability of two hierarchical grouping techniques case I: Sensitivity to data errors. *Journal of the American Statistical Association*, *69*(346), 440-445. Retrieved from http://dx.doi.org/10.1080/01621459.1974.10482971 doi: 10.1080/01621459.1974.10482971

Balan, R. K. (2006). *Simplifying cyber foraging* (Doctoral dissertation, Pittsburgh, PA, USA). Retrieved from http://dl.acm.org/citation.cfm?id=1195029 (AAI3222544)

Balan, R. K., Satyanarayanan, M., Park, S. Y. & Okoshi, T. (2003). Tactics-based remote execution for mobile computing. In *Proceedings of the 1st international conference on mobile systems, applications*

and services (pp. 273–286). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1066116.1066125 doi: 10.1145/1066116.1066125

Ball, T. & Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, *16*(4), 1319–1360. Retrieved from http://doi.acm.org/10.1145/183432.183527 doi: 10.1145/183432.183527

Batalas, N. & Markopoulos, P. (2012). Considerations for computerized in situ data collection platforms. In *Proceedings of the 4th ACM SIGCHI symposium on engineering interactive computing systems* (pp. 231–236). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2305484.2305522 doi: 10.1145/2305484.2305522

Beckman, P., Iskra, K., Yoshii, K. & Coghlan, S. (2006). The influence of operating systems on the performance of collective operations at extreme scale. In *2006 IEEE international conference on cluster computing* (p. 1-12). doi: 10.1109/CLUSTR.2006.311846

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S. & Nataraj, A. (2008). Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, *11*(1), 3–16. Retrieved from http://dx.doi.org/10.1007/s10586-007-0047-2 doi: 10.1007/s10586-007-0047-2

Berry, H., Gracia Perez, D. & Temam, O. (2006). Chaos in computer performance. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *16*(1). Retrieved from http://scitation.aip.org/content/aip/journal/chaos/16/1/10.1063/1.2159147 doi: http://dx.doi.org/10.1063/1.2159147

Breunig, M. M., Kriegel, H.-P., Ng, R. T. & Sander, J. (2000). LOF: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on management of data* (pp. 93–104). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/342009.335388 doi: 10.1145/342009.335388

Brouwers, N. & Langendoen, K. (2012). Pogo, a middleware for mobile phone sensing. In *Proceedings of the 13th international middleware conference* (pp. 21–40). New York, NY, USA: Springer-Verlag New York, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2442626.2442629

Brunette, W., Sundt, M., Dell, N., Chaudhri, R., Breit, N. & Borriello, G. (2013). Open data kit 2.0: Expanding and refining information services for developing regions. In *Proceedings of the 14th workshop on mobile computing systems and applications* (pp. 10:1–10:6). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2444776.2444790 doi: 10.1145/2444776.2444790

Brys, G., Hubert, M. & Struyf, A. (2004). A robust measure of skewness. *Journal of Computational and Graphical Statistics*, *13*(4). doi: 10.1198/106186004X12632

Cadar, C., Dunbar, D. & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on operating systems design and implementation* (pp. 209–224). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1855741.1855756

Cadar, C. & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Communications of the ACM*, *56*(2), 82–90. Retrieved from http://doi.acm.org/10.1145/2408776.2408795 doi: 10.1145/2408776.2408795

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-8*(6), 679-698. doi: 10.1109/TPAMI.1986.4767851

Cathey, R. J., Jensen, E. C., Beitzel, S. M., Frieder, O. & Grossman, D. (2007). Exploiting parallelism to support scalable hierarchical clustering. *Journal of the American Society for Information Science and Technology*, *58*(8), 1207–1221. Retrieved from http://dx.doi.org/10.1002/asi.20596 doi: 10.1002/asi.20596

Chan, A., Gao, C. & Rau-Chaplin, A. (2005). A coarse grained parallel algorithm for closest larger ancestors in trees with applications to single link clustering. In L. T. Yang, O. F. Rana, B. Di Martino & J. Dongarra (Eds.), *High performance computing and communications: First international conference, HPCC 2005, sorrento, italy, september 21-23, 2005. proceedings* (pp. 856–865). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/11557654_96 doi: 10.1007/11557654_96

Chang, D. J., Desoky, A. H., Ouyang, M. & Rouchka, E. C. (2009). Compute pairwise manhattan distance and pearson correlation coefficient of data points with GPU. In *Software engineering, artificial intelligences, networking and parallel/distributed computing, 2009. SNPD '09. 10th ACIS international conference on* (p. 501-506). doi: 10.1109/SNPD.2009.34

Chang, D.-J., Kantardzic, M. M. & Ouyang, M. (2009). Hierarchical clustering with CUDA/GPU. In *Symposium on computer animation* (p. 7-12).

Chang, Y.-S. & Hung, S.-H. (2011). Developing collaborative applications with mobile cloud - a case study of speech recognition. *Journal of Internet Services and Information Security (JISIS)*, *1*(1), 18–36.

Chen, G., Kang, B.-T., Kandemir, M., Vijaykrishnan, N., Irwin, M. J. & Chandramouli, R. (2004). Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices. *IEEE Transactions on Parallel and Distributed Systems*, *15*(9), 795–809. Retrieved from http://dx.doi.org/10.1109/TPDS.2004.47 doi: 10.1109/TPDS.2004.47

Chen, X., Jiao, L., Li, W. & Fu, X. (2016). Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, *24*(5), 2795-2808. doi: 10.1109/TNET.2015.2487344

Chipounov, V., Kuznetsov, V. & Candea, G. (2011). S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems* (pp. 265–278). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1950365.1950396 doi: 10.1145/1950365.1950396

Chipounov, V., Kuznetsov, V. & Candea, G. (2012). The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, *30*(1), 2:1–2:49. Retrieved from http://doi.acm.org/10.1145/2110356.2110358 doi: 10.1145/2110356.2110358

Chu, H.-h., Song, H., Wong, C., Kurakake, S. & Katagiri, M. (2004). Roam, a seamless application framework. *Journal of Systems and Software*, *69*(3), 209–226. Retrieved from http://dx.doi.org/10.1016/S0164-1212(03)00052-9 doi: 10.1016/S0164-1212(03)00052-9

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M. & Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on computer systems* (pp. 301–314).

New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1966445.1966473 doi: 10.1145/1966445.1966473

Cidon, A., London, T. M., Katti, S., Kozyrakis, C. & Rosenblum, M. (2011). MARS: adaptive remote execution for multi-threaded mobile devices. In *Proceedings of the 3rd ACM SOSP workshop on networking, systems, and applications on mobile handhelds* (pp. 1:1–1:6). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2043106.2043107 doi: 10.1145/2043106.2043107

Cleven, A., Gubler, P. & Hüner, K. M. (2009). Design alternatives for the evaluation of design science research artifacts. In *Proceedings of the 4th international conference on design science research in information systems and technology* (pp. 19:1–19:8). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1555619.1555645 doi: 10.1145/1555619.1555645

Colin, A. & Puaut, I. (2000). Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, *18*(2), 249–274. Retrieved from http://dx.doi.org/10.1023/A:1008149332687 doi: 10.1023/A:1008149332687

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R. & Bahl, P. (2010). MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on mobile systems, applications, and services* (pp. 49–62). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1814433.1814441 doi: 10.1145/1814433.1814441

Dash, M., Petrutiu, S. & Scheuermann, P. (2007). pPOP: Fast yet accurate parallel hierarchical clustering using partitioning. *Data & Knowledge Engineering*, *61*(3), 563-578. Retrieved from http://www.sciencedirect.com/science/article/pii/S0169023X06001340 doi: http://dx.doi.org/10.1016/j.datak.2006.07.004

Dash, M., Tan, K. L. & Liu, H. (2001). Efficient yet accurate clustering. In *Data mining, 2001. ICDM 2001, proceedings IEEE international conference on* (p. 99-106). doi: 10.1109/ICDM.2001.989506

Daubechies, I. et al. (1992). *Ten lectures on wavelets* (Vol. 61). SIAM.

Day, W. H. E. & Edelsbrunner, H. (1984). Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, *1*(1), 7–24. Retrieved from http://dx.doi.org/10.1007/BF01890115 doi: 10.1007/BF01890115

De, P., Kothari, R. & Mann, V. (2007). Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proceedings of the 2007 IEEE international conference on cluster computing* (pp. 331–340). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/CLUSTR.2007.4629247 doi: 10.1109/CLUSTR.2007.4629247

De, P. & Mann, V. (2010). jitSim: A simulator for predicting scalability of parallel applications in presence of OS jitter. In *Proceedings of the 16th international Euro-Par conference on parallel processing: Part I* (pp. 117–130). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1887695.1887709

De, P., Mann, V. & Mittaly, U. (2009). Handling OS jitter on multicore multithreaded systems. In *Parallel distributed processing, 2009. IPDPS 2009. IEEE international symposium on* (p. 1-12). doi: 10.1109/IPDPS.2009.5161046

Defays, D. (1977). An efficient algorithm for a complete link method. *The Computer Journal*, *20*(4), 364-366. Retrieved from http://comjnl.oxfordjournals.org/content/20/4/364.abstract doi: 10.1093/comjnl/20.4.364

Dou, A., Kalogeraki, V., Gunopulos, D., Mielikainen, T. & Tuulos, V. H. (2010). Misco: A mapreduce framework for mobile systems. In *Proceedings of the 3rd international conference on pervasive technologies related to assistive environments* (pp. 32:1–32:8). New York, NY, USA: ACM. Retrieved from http://doi.acm.org.elibrary.jcu.edu.au/10.1145/1839294.1839332 doi: 10.1145/1839294.1839332

Du, Z. & Lin, F. (2004). A hierarchical clustering algorithm for MIMD architecture. *Computational Biology and Chemistry*, *28*(5-6), 417-419. Retrieved from http://www.sciencedirect.com/science/article/pii/S1476927104000659 doi: http://dx.doi.org/10.1016/j.compbiolchem.2004.09.002

Du, Z. & Lin, F. (2005). A novel parallelization approach for hierarchical clustering. *Parallel Computing*, *31*(5), 523-527. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167819105000256 doi: http://dx.doi.org/10.1016/j.parco.2005.01.001

Duga, N. (2011). *Optimality analysis and middleware design for heterogeneous cloud HPC in mobile devices* (Unpublished doctoral dissertation). Citeseer.

Eler, M. M., Endo, A. T. & Durelli, V. H. (2016). An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective. *Journal of Systems and Software*, *121*, 281 - 297. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121216000868 doi: http://dx.doi.org/10.1016/j.jss.2016.03.020

El-Hamdouchi, A. & Willett, P. (1989). Comparison of hierarchic agglomerative clustering methods for document retrieval. *The Computer Journal*, *32*(3), 220-227. Retrieved from http://comjnl.oxfordjournals.org/content/32/3/220.abstract doi: 10.1093/comjnl/32.3.220

Emoto, M., Ohdachi, S., Watanabe, K., Sudo, S. & Nagayama, Y. (2006). Server for experimental data from LHD. *Fusion Engineering and Design*, *81*(15-17), 2019 - 2023. Retrieved from http://www.sciencedirect.com/science/article/pii/S0920379606001165 (5th IAEA TM on Control, Data Acquisition, and Remote Participation for Fusion Research 5th IAEA TM 5th IAEA TM on Control, Data Acquisition and Remote Participation for Fusion Research) doi: http://dx.doi.org/10.1016/j.fusengdes.2006.04.030

Endo, Y., Wang, Z., Chen, J. B. & Seltzer, M. (1996). Using latency to evaluate interactive system performance. In *Proceedings of the second USENIX symposium on operating systems design and implementation* (pp. 185–199). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/238721.238775 doi: 10.1145/238721.238775

Engblom, J. (2002). Processor pipelines and static worst-case execution time analysis.

Ernst, R. & Ye, W. (1997). Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the 1997 IEEE/ACM international conference on computer-aided design* (pp. 598–604). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=266388.266562

Falaki, H., Mahajan, R. & Estrin, D. (2011). SystemSens: A tool for monitoring usage in smartphone

research deployments. In *Proceedings of the sixth international workshop on MobiArch* (pp. 25–30). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1999916.1999923 doi: 10.1145/1999916.1999923

Feng, Z., Zhou, B. & Shen, J. (2007). A parallel hierarchical clustering algorithm for PCs cluster system. *Neurocomputing*, *70*(4-6), 809-818. Retrieved from http://www.sciencedirect.com/science/article/pii/S0925231206002864 (Advanced Neurocomputing Theory and Methodology: Selected papers from the International Conference on Intelligent Computing 2005 (ICIC 2005)) doi: http://dx.doi.org/10.1016/j.neucom.2006.10.034

Fernando, N., Loke, S. W. & Rahayu, W. (2013a). Honeybee: A programming framework for mobile crowd computing. In K. Zheng, M. Li & H. Jiang (Eds.), *Mobile and ubiquitous systems: Computing, networking, and services: 9th international conference, MobiQuitous 2012, Beijing, China, december 12-14, 2012. Revised selected papers* (pp. 224–236). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-40238-8_19 doi: 10.1007/978-3-642-40238-8_19

Fernando, N., Loke, S. W. & Rahayu, W. (2013b). Mobile cloud computing: A survey. *Future Generation Computing Systems*, *29*(1), 84–106. Retrieved from http://dx.doi.org/10.1016/j.future.2012.05.023 doi: 10.1016/j.future.2012.05.023

Ferrari, A., Giordano, S. & Puccinelli, D. (2016). Reducing your local footprint with anyrun computing. *Computer Communications*, *81*, 1 - 11. Retrieved from http://www.sciencedirect.com/science/article/pii/S0140366416000293 doi: http://dx.doi.org/10.1016/j.comcom.2016.01.006

Ferreira, K. B., Bridges, P. & Brightwell, R. (2008). Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on supercomputing* (pp. 19:1–19:12). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=1413370.1413390

Fielding, R. T. & Taylor, R. N. (2000). Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on software engineering* (pp. 407–416). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/337180.337228 doi: 10.1145/337180.337228

Flinn, J., Park, S. & Satyanarayanan, M. (2002). Balancing performance, energy, and quality in pervasive computing. In *Proceedings of the 22nd international conference on distributed computing systems (ICDCS'02)* (pp. 217–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=850928.851899

Flores, H., Hui, P., Tarkoma, S., Li, Y., Srirama, S. & Buyya, R. (2015). Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, *53*(3), 80-88. doi: 10.1109/MCOM.2015.7060486

Flores, H. & Srirama, S. (2013). Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning. In *Proceeding of the fourth ACM workshop on mobile cloud computing and services* (pp. 9–16). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2482981.2482984 doi: 10.1145/2482981.2482984

Flores, H. & Srirama, S. N. (2014). Mobile cloud middleware. *Journal of Systems and Soft-*

ware, *92*, 82 - 94. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121213002318 doi: http://dx.doi.org/10.1016/j.jss.2013.09.012

Fowlkes, E. B. & Mallows, C. L. (1983). A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, *78*(383), 553-569. Retrieved from http://amstat.tandfonline.com/doi/abs/10.1080/01621459.1983.10478008 doi: 10.1080/01621459.1983.10478008

Franke, B. (2008). Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on software & compilers for embedded systems* (pp. 69–78). New York, NY, USA: ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1361096.1361109

Froehlich, J., Chen, M. Y., Consolvo, S., Harrison, B. & Landay, J. A. (2007). MyExperience: A system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of the 5th international conference on mobile systems, applications and services* (pp. 57–70). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1247660.1247670 doi: 10.1145/1247660.1247670

Gao, B., He, L., Liu, L., Li, K. & Jarvis, S. A. (2012). From mobiles to clouds: Developing energy-aware offloading strategies for workflows. In *Proceedings of the 2012 ACM/IEEE 13th international conference on grid computing* (pp. 139–146). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/Grid.2012.20 doi: 10.1109/Grid.2012.20

Gao, W., Li, Y., Lu, H., Wang, T. & Liu, C. (2014). On exploiting dynamic execution patterns for workload offloading in mobile cloud applications. In *2014 IEEE 22nd international conference on network protocols* (p. 1-12). doi: 10.1109/ICNP.2014.22

Garg, R. & De, P. (2006). Impact of noise on scaling of collectives: An empirical evaluation. In Y. Robert, M. Parashar, R. Badrinath & V. Prasanna (Eds.), *High performance computing - HiPC 2006* (Vol. 4297, p. 460-471). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/11945918_45 doi: 10.1007/11945918_45

Gartner, Inc. (2014). *Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013.* [Press release]. Retrieved 2014-04-22, from http://www.gartner.com/newsroom/id/2665715

Gartner, Inc. (2016a). *Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015.* [Press release]. Retrieved 2016-12-01, from http://www.gartner.com/newsroom/id/3165317

Gartner, Inc. (2016b). *Gartner says worldwide smartphone sales grew 9.7 percent in fourth quarter of 2015.* [Press release]. Retrieved 2016-12-01, from http://www.gartner.com/newsroom/id/3215217

Geng, Y., Hu, W., Yang, Y., Gao, W. & Cao, G. (2015). Energy-efficient computation offloading in cellular networks. In *2015 ieee 23rd international conference on network protocols (ICNP)* (p. 145-155). doi: 10.1109/ICNP.2015.20

Gil, B. & Trezentos, P. (2011). Impacts of data interchange formats on energy consumption and performance in smartphones. In *Proceedings of the 2011 workshop on open source and design of communication* (pp. 1–6). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2016716.2016718 doi: 10.1145/2016716.2016718

Gil, J. Y., Lenz, K. & Shimron, Y. (2011). A microbenchmark case study and lessons learned. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE'11, AOOPES'11, NEAT'11, & VMIL'11* (pp. 297–308). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2095050.2095100 doi: 10.1145/2095050.2095100

Giurgiu, I., Riva, O. & Alonso, G. (2012). Dynamic software deployment from clouds to mobile devices. In *Proceedings of the 13th international middleware conference* (pp. 394–414). New York, NY, USA: Springer-Verlag New York, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2442626.2442651

Giurgiu, I., Riva, O., Juric, D., Krivulev, I. & Alonso, G. (2009). Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Proceedings of the ACM/IFIP/USENIX 10th international conference on middleware* (pp. 83–102). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1813355.1813362

Giusto, P., Martin, G. & Harcourt, E. (2001). Reliable estimation of execution time of embedded software. In *Proceedings of the conference on design, automation and test in Europe* (pp. 580–589). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=367072.367827

Google Inc. (2016). *Android NDK*. Retrieved 2016-09-28, from https://developer.android.com/ndk/index.html

Google, Inc. (2016). *Top grossing apps - Android apps on Google Play*. Retrieved 2016-11-14, from https://play.google.com/store/apps/collection/topgrossing?hl=en

Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M. & Chen, X. (2012). COMET: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX symposium on operating systems design and implementation (OSDI 12)* (pp. 93–106). Hollywood, CA: USENIX. Retrieved from https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon

Goyal, S. & Carter, J. (2004). A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Mobile computing systems and applications, 2004. WMCSA 2004. sixth IEEE workshop on* (p. 186-195). doi: 10.1109/MCSA.2004.2

Graham, S. L., Kessler, P. B. & Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Notices*, *17*(6), 120–126. Retrieved from http://doi.acm.org/10.1145/872726.806987 doi: 10.1145/872726.806987

Gregor, S. & Hevner, A. R. (2013). Positioning and presenting design science research for maximum impact. *MIS Quarterly*, *37*(2), 337–356. Retrieved from http://dl.acm.org/citation.cfm?id=2535658.2535660

Gu, X., Nahrstedt, K., Messer, A., Greenberg, I. & Milojicic, D. (2003). Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proceedings of the first IEEE international conference on pervasive computing and communications* (pp. 107–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=826025.826367

Gu, X., Nahrstedt, K., Messer, A., Greenberg, I. & Milojicic, D. (2004). Adaptive offloading for pervasive

computing. *IEEE Pervasive Computing*, *3*(3), 66-73. doi: 10.1109/MPRV.2004.1321031

Gubbi, J., Buyya, R., Marusic, S. & Palaniswami, M. (2013). Internet of things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, *29*(7), 1645 - 1660. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167739X13000241 (Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications - Big Data, Scalable Analytics, and Beyond) doi: http://dx.doi.org/10.1016/j.future.2013.01.010

Guha, S., Rastogi, R. & Shim, K. (1998). CURE: An efficient clustering algorithm for large databases. In *Proceedings of the 1998 ACM SIGMOD international conference on management of data* (pp. 73–84). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/276304.276312 doi: 10.1145/276304.276312

Gurun, S., Krintz, C. & Wolski, R. (2008). NWSLite: A general-purpose, nonparametric prediction utility for embedded systems. *ACM Transactions on Embedded Computing Systems*, *7*(3), 32:1–32:36. Retrieved from http://doi.acm.org/10.1145/1347375.1347385 doi: 10.1145/1347375.1347385

Haahr, M. (2016). *Random.org: true random number service.* Retrieved 2016-10-24, from http://www.random.org

Hadjidoukas, P. E. & Amsaleg, L. (2008). Parallelization of a hierarchical data clustering algorithm using OpenMP. In M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony & M. Voss (Eds.), *OpenMP shared memory parallel programming: International workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12-15, 2006. proceedings* (pp. 289–299). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-68555-5_24 doi: 10.1007/978-3-540-68555-5_24

Han, S., Zhang, S., Cao, J., Wen, Y. & Zhang, Y. (2008). A resource aware software partitioning algorithm based on mobility constraints in pervasive grid environments. *Future Generation Computer Systems*, *24*(6), 512 - 529. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167739X07001306 doi: http://dx.doi.org/10.1016/j.future.2007.07.013

Harmon, M. G., Baker, T. P. & Whalley, D. B. (1994). A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, *7*(2), 159–182. Retrieved from http://dx.doi.org/10.1007/BF01088803 doi: 10.1007/BF01088803

Hendrix, W., Palsetia, D., Patwary, M. M. A., Agrawal, A., Liao, W.-k. & Choudhary, A. N. (2013). A scalable algorithm for single-linkage hierarchical clustering on distributed-memory architectures. In *IEEE symposium on large data analysis and visualization* (pp. 7–13).

Hendrix, W., Patwary, M. M. A., Agrawal, A., k. Liao, W. & Choudhary, A. (2012). Parallel hierarchical clustering on shared memory platforms. In *High performance computing (HiPC), 2012 19th international conference on* (p. 1-9). doi: 10.1109/HiPC.2012.6507511

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105. Retrieved from http://dl.acm.org/citation.cfm?id=2017212.2017217

Hill, M. D. & Marty, M. R. (2008). Amdahl's law in the multicore era. *IEEE Computer*, *41*, 33-38.

Hoberock, J. & Bell, N. (2016). *Thrust documentation: extrema.* Retrieved 2016-11-09, from `https://thrust.github.io/doc/group__extrema.html`

Hoefler, T., Schneider, T. & Lumsdaine, A. (2010). Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis* (pp. 1–11). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/SC.2010.12` doi: 10.1109/SC.2010.12

Holzer, A., Januzaj, V., Kugele, S. & Tautschnig, M. (2010). Timely time estimates. In *Proceedings of the 4th international conference on leveraging applications of formal methods, verification, and validation - volume part I* (pp. 33–46). Berlin, Heidelberg: Springer-Verlag. Retrieved from `http://dl.acm.org/citation.cfm?id=1939281.1939287`

Hong, Y. J., Kumar, K. & Lu, Y. H. (2009). Energy efficient content-based image retrieval for mobile systems. In *2009 IEEE international symposium on circuits and systems* (p. 1673-1676). doi: 10.1109/ISCAS.2009.5118095

Hoseini-Tabatabaei, S. A., Gluhak, A. & Tafazolli, R. (2013). A survey on smartphone-based systems for opportunistic user context recognition. *ACM Computing Surveys*, *45*(3), 27:1–27:51. Retrieved from `http://doi.acm.org/10.1145/2480741.2480744` doi: 10.1145/2480741.2480744

Huang, D., Zhang, X., Kang, M. & Luo, J. (2010). MobiCloud: Building secure cloud framework for mobile computing and communication. In *Service oriented system engineering (SOSE), 2010 fifth IEEE international symposium on* (p. 27-34). doi: 10.1109/SOSE.2010.20

Huang, L., Jia, J., Yu, B., gon Chun, B., Maniatis, P. & Naik, M. (2010). Predicting execution time of computer programs using sparse polynomial regression. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel & A. Culotta (Eds.), *Advances in neural information processing systems 23* (pp. 883–891). Curran Associates, Inc. Retrieved from `http://papers.nips.cc/paper/4145-predicting-execution-time-of-computer-programs-using-sparse-polynomial-regression.pdf`

Hubert, M. & Vandervieren, E. (2008). An adjusted boxplot for skewed distributions. *Computational Statistics & Data Analysis*, *52*(12), 5186–5201. Retrieved from `http://dx.doi.org/10.1016/j.csda.2007.11.008` doi: 10.1016/j.csda.2007.11.008

Huerta-Canepa, G. & Lee, D. (2010). A virtual cloud computing provider for mobile devices. In *Proceedings of the 1st ACM workshop on mobile cloud computing & services: Social networks and beyond* (pp. 6:1–6:5). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org.elibrary.jcu.edu.au/10.1145/1810931.1810937` doi: 10.1145/1810931.1810937

Hung, S.-H., Shieh, J.-P. & Lee, C.-P. (2012). Migrating android applications to the cloud. *Applications and Developments in Grid, Cloud, and High Performance Computing*, 307.

Hung, S.-H., Shih, C.-S., Shieh, J.-P., Lee, C.-P. & Huang, Y.-H. (2012). Executing mobile applications on the cloud: Framework and issues. *Computers & Mathematics with Applications*, *63*(2), 573 - 587. Retrieved from `http://www.sciencedirect.com/science/article/pii/S0898122111009084` (Advances in context, cognitive, and secure computing) doi: http://dx.doi.org/10.1016/j.camwa.2011.10.044

Hunt, G. C. & Scott, M. L. (1999). The coign automatic distributed partitioning system. In *Proceedings of the third symposium on operating systems design and implementation* (pp. 187–200). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=296806.296826

Igler, B. (2013). Feature evaluation for mobile applications: A design science approach based on evolutionary software prototypes. In *Proceedings of the second international conference on design, user experience, and usability: Web, mobile, and product design* (pp. 673–681). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-39253-5_75 doi: 10.1007/978-3-642-39253-5_75

Imai, S. (2012). *Task offloading between smartphones and distributed computational resources* (Unpublished doctoral dissertation). Rensselaer Polytechnic Institute.

Imai, S. & Varela, C. A. (2011). Light-weight adaptive task offloading from smartphones to nearby computational resources. In *Proceedings of the 2011 ACM symposium on research in applied computation* (pp. 146–152). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2103380.2103411 doi: 10.1145/2103380.2103411

International Data Corporation. (2013a). *App Annie & IDC portable gaming report: iOS App Store, Google Play games combined eclipsed dedicated handheld games in consumer spending in Q4 2012.* [Press release]. Retrieved 2014-04-22, from http://www.idc.com/getdoc.jsp?containerId=prUS23959813

International Data Corporation. (2013b). *Nearly 80% of manufacturers to develop mobile application this year, according to IDC manufacturing insights.* [Press release]. Retrieved 2014-04-22, from http://www.idc.com/getdoc.jsp?containerId=prUS24240213

International Data Corporation. (2014a). *New IDC MarketScape presents a vendor assessment of providers offering mobile application development, testing and infrastructure services.* [Press release]. Retrieved 2014-04-22, from http://www.idc.com/getdoc.jsp?containerId=prUS24778414

International Data Corporation. (2014b). *Worldwide smartphone shipments top one billion units for the first time, according to IDC.* [Press release]. Retrieved 2014-04-22, from http://www.idc.com/getdoc.jsp?containerId=prUS24645514

Jain, A. K. & Dubes, R. C. (1988). *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=46712

Jain, A. K., Murty, M. N. & Flynn, P. J. (1999). Data clustering: A review. *ACM Computing Surveys*, *31*(3), 264–323. Retrieved from http://doi.acm.org/10.1145/331499.331504 doi: 10.1145/331499.331504

Jamal, M. H. & Waheed, A. (2008). Precise measurement of execution time of concurrent, symmetric, and short tasks. In *34th international computer measurement group conference, December 7-12, 2008, Las Vegas, Nevada, USA, proceedings* (pp. 149–160). Retrieved from http://dblp.uni-trier.de/rec/bibtex/conf/cmg/Jamal08

Januzaj, V., Mauersberger, R. & Biechele, F. (2009). Performance modelling for avionics systems. In R. Moreno-Díaz, F. Pichler & A. Quesada-Arencibia (Eds.), *Computer aided systems theory - EUROCAST 2009: 12th international conference, Las Palmas de Gran Canaria, Spain, Feb-*

ruary 15-20, 2009, revised selected papers (pp. 833–840). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-04772-5_107 doi: 10.1007/978-3-642-04772-5_107

Jenkins, S. P. & Van Kerm, P. (2006). Trends in income inequality, pro-poor income growth, and income mobility. *Oxford Economic Papers*, *58*(3), 531-548. Retrieved from http://oep.oxfordjournals.org/content/58/3/531.abstract doi: 10.1093/oep/gpl014

Jeon, Y. & Yoon, S. (2015). Multi-threaded hierarchical clustering by parallel nearest-neighbor chaining. *IEEE Transactions on Parallel and Distributed Systems*, *26*(9), 2534-2548. doi: 10.1109/TPDS.2014.2355205

Jin, C., Liu, R., Chen, Z., Hendrix, W., Agrawal, A. & Choudhary, A. (2015). A scalable hierarchical clustering algorithm using Spark. In *Big data computing service and applications (BigDataService), 2015 IEEE first international conference on* (p. 418-426). doi: 10.1109/BigDataService.2015.67

Kakadia, D., Saripalli, P. & Varma, V. (2013). MECCA: Mobile, efficient cloud computing workload adoption framework using scheduler customization and workload migration decisions. In *Proceedings of the first international workshop on mobile cloud computing & networking* (pp. 41–46). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2492348.2492357 doi: 10.1145/2492348.2492357

Kaufman, L. & Rousseeuw, P. J. (1990). *Finding groups in data: an introduction to cluster analysis*. New York, USA: John Wiley & Sons. doi: 10.1002/9780470316801

Kemp, R., Palmer, N., Kielmann, T. & Bal, H. (2010). Cuckoo: A computation offloading framework for smartphones. In *Proceedings of the 2nd international conference on mobile computing, applications, and services* (pp. 59–79). doi: 10.1007/978-3-642-29336-8_4

Khronos Group. (2016). *OpenCL - the open standard for parallel programming of heterogeneous systems.* Retrieved 2016-11-09, from https://www.khronos.org/opencl/

Kim, J.-M. & Kim, J.-S. (2012). Androbench: Benchmarking the storage performance of android-based mobile devices. In S. Sambath & E. Zhu (Eds.), *Frontiers in computer education* (Vol. 133, p. 667-674). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-27552-4_89 doi: 10.1007/978-3-642-27552-4_89

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385–394. Retrieved from http://doi.acm.org/10.1145/360248.360252 doi: 10.1145/360248.360252

Kohlhoff, K. J., Sosnick, M. H., Hsu, W. T., Pande, V. S. & Altman, R. B. (2011). CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, *27*(16), 2321-2322. Retrieved from http://bioinformatics.oxfordjournals.org/content/27/16/2321.abstract doi: 10.1093/bioinformatics/btr386

Kosta, S., Aucinas, A., Hui, P., Mortier, R. & Zhang, X. (2012). ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 proceedings IEEE* (p. 945-953). doi: 10.1109/INFCOM.2012.6195845

Kotker, J., Sadigh, D. & Seshia, S. A. (2011). Timing analysis of interrupt-driven programs under context bounds. In *Proceedings of the international conference on formal methods in computer-aided design*

(pp. 81–90). Austin, TX: FMCAD Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2157654.2157670

Kovachev, D., Cao, Y. & Klamma, R. (2011). Mobile cloud computing: A comparison of application models. *CoRR*, *abs/1107.4940*. Retrieved from http://arxiv.org/abs/1107.4940

Kovachev, D., Cao, Y. & Klamma, R. (2012). Augmenting pervasive environments with an XMPP-based mobile cloud middleware. In M. Gris & G. Yang (Eds.), *Mobile computing, applications, and services: Second international ICST conference, MobiCASE 2010, Santa Clara, CA, USA, October 25-28, 2010, revised selected papers* (pp. 361–372). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-29336-8_25 doi: 10.1007/978-3-642-29336-8_25

Kovachev, D. & Klamma, R. (2012). Framework for computation offloading in mobile cloud computing. *International Journal of Interactive Multimedia and Artificial Intelligence*, *1*(7), 6-15. Retrieved from http://www.ijimai.org/journal/sites/default/files/files/2012/11/ijimai20121_7_1_pdf_62175.pdf doi: 10.9781/ijimai.2012.171

Kremer, U., Hicks, J. & Rehg, J. (2003). A compilation framework for power and energy management on mobile computers. In H. G. Dietz (Ed.), *Languages and compilers for parallel computing: 14th international workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1–3, 2001 revised papers* (pp. 115–131). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-35767-X_8 doi: 10.1007/3-540-35767-X_8

Kristensen, M. D. (2010). Scavenger: Transparent development of efficient cyber foraging applications. In *Pervasive computing and communications (PerCom), 2010 IEEE international conference on* (p. 217-226). doi: 10.1109/PERCOM.2010.5466972

Kumar, K., Liu, J., Lu, Y.-H. & Bhargava, B. (2013). A survey of computation offloading for mobile systems. *Mobile Network Applications*, *18*(1), 129–140. Retrieved from http://dx.doi.org/10.1007/s11036-012-0368-0 doi: 10.1007/s11036-012-0368-0

Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B.-G., ... Paek, Y. (2013). Mantis: Automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX conference on annual technical conference* (pp. 297–308). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org.elibrary.jcu.edu.au/citation.cfm?id=2535461.2535498

Kwon, Y.-W. & Tilevich, E. (2013). Reducing the energy consumption of mobile applications behind the scenes. In *ICSM* (pp. 170–179).

Langfelder, P., Zhang, B. & Horvath, S. (2008). Defining clusters from a hierarchical cluster tree: the dynamic tree cut package for R. *Bioinformatics*, *24*(5), 719-720. Retrieved from http://bioinformatics.oxfordjournals.org/content/24/5/719.abstract doi: 10.1093/bioinformatics/btm563

Lattner, C. (2008). LLVM and Clang: Next generation compiler technology. In *The BSD conference* (pp. 1–2).

Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

Levin, J. (2012). *Mac OS X and iOS Internals: To the Apple's Core*. John Wiley & Sons.

Lewis, G. & Lago, P. (2015). Architectural tactics for cyber-foraging: Results of a systematic literature review. *Journal of Systems and Software*, *107*, 158 - 186. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121215001211 doi: http://dx.doi.org/10.1016/j.jss.2015.06.005

Li, Z., Li, K., Xiao, D. & Yang, L. (2007). An adaptive parallel hierarchical clustering algorithm. In R. Perrott, B. M. Chapman, J. Subhlok, R. F. de Mello & L. T. Yang (Eds.), *High performance computing and communications: Third international conference, HPCC 2007, Houston, USA, September 26-28, 2007. proceedings* (pp. 97–107). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-75444-2_15 doi: 10.1007/978-3-540-75444-2_15

Li, Z., Wang, C. & Xu, R. (2001). Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on compilers, architecture, and synthesis for embedded systems* (pp. 238–246). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/502217.502257 doi: 10.1145/502217.502257

Li, Z. & Xu, R. (2002). Energy impact of secure computation on a handheld device. In *Workload characterization, 2002. WWC-5. 2002 IEEE international workshop on* (p. 109-117). doi: 10.1109/WWC.2002.1226499

Liu, J. & Lu, Y.-H. (2010). Energy savings in privacy-preserving computation offloading with protection by homomorphic encryption. In *Proceedings of the 2010 international conference on power aware computing and systems, hotpower* (Vol. 10, pp. 1–7).

Ma, R. K. K., Lam, K. T. & Wang, C. L. (2011). eXCloud: Transparent runtime support for scaling mobile applications in cloud. In *Cloud and service computing (CSC), 2011 international conference on* (p. 103-110). doi: 10.1109/CSC.2011.6138505

Ma, R. K. K. & Wang, C. L. (2012). Lightweight application-level task migration for mobile cloud computing. In *2012 IEEE 26th international conference on advanced information networking and applications* (p. 550-557). doi: 10.1109/AINA.2012.124

Malhat, M. G. & El-Sisi, A. B. (2015). Parallel ward clustering for chemical compounds using OpenCL. In *Computer engineering systems (ICCES), 2015 tenth international conference on* (p. 23-27). doi: 10.1109/ICCES.2015.7393011

March, V., Gu, Y., Leonardi, E., Goh, G., Kirchberg, M. & Lee, B. S. (2011). µcloud: Towards a new paradigm of rich mobile applications. *Procedia Computer Science*, *5*, 618 - 624. Retrieved from http://www.sciencedirect.com/science/article/pii/S1877050911004054 doi: http://dx.doi.org/10.1016/j.procs.2011.07.080

Marinelli, E. E. (2009). *Hyrax: cloud computing on mobile devices using MapReduce* (Unpublished doctoral dissertation). Pittsburgh, PA, USA.

Matthews, J., Chang, M., Feng, Z., Srinivas, R. & Gerla, M. (2011). Powersense: Power aware dengue diagnosis on mobile phones. In *Proceedings of the first acm workshop on mobile systems, applications, and services for healthcare* (pp. 6:1–6:6). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2064942.2064951 doi: 10.1145/2064942.2064951

Merrill, D. & Grimshaw, A. (2011). High performance and scalable radix sorting: A case study of

implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, *21*(02), 245-272. Retrieved from http://www.worldscientific.com/doi/abs/10.1142/S0129626411000187 doi: 10.1142/S0129626411000187

Messer, A., Greenberg, I., Bernadat, P., Milojicic, D., Chen, D., Giuli, T. J. & Gu, X. (2002). Towards a distributed platform for resource-constrained devices. In *Proceedings of the 22nd international conference on distributed computing systems (ICDCS'02)* (pp. 43–). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=850928.851826

Messinger, D. & Lewis, G. A. (2013). Application virtualizaton as a strategy for cyber foraging in resource-constrained environments. *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Note CMU/SEI-2013-TN-007*.

Microsoft Corporation. (2016). *Intro to the Universal Windows Platform.* Retrieved 2016-09-28, from https://msdn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

Mills, D. L. (2010). *Computer network time synchronization: The network time protocol on earth and in space, second edition* (2nd ed.). Boca Raton, FL, USA: CRC Press, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1951866

Mittal, S. (2014). A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer Aided Engineering and Technology*, *6*(4), 440-459. Retrieved from http://www.inderscienceonline.com/doi/abs/10.1504/IJCAET.2014.065419 doi: 10.1504/IJCAET.2014.065419

Mohapatra, S. & Venkatasubramanian, N. (2003). Optimizing power using a reconfigurable middleware. *UC Irvine*.

Morari, A., Gioiosa, R., Wisniewski, R., Cazorla, F. & Valero, M. (2011). A quantitative analysis of OS noise. In *Parallel distributed processing symposium (IPDPS), 2011 IEEE international* (p. 852-863). doi: 10.1109/IPDPS.2011.84

Murtagh, F. (1983). A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, *26*(4), 354-359. Retrieved from http://comjnl.oxfordjournals.org/content/26/4/354.abstract doi: 10.1093/comjnl/26.4.354

Murtagh, F. (1985). Multidimensional clustering algorithms. *Compstat Lectures, Vienna: Physika Verlag, 1985*.

Nataraj, A., Morris, A., Malony, A. D., Sottile, M. & Beckman, P. (2007). The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE conference on supercomputing* (pp. 29:1–29:12). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1362622.1362662 doi: 10.1145/1362622.1362662

Ni, T. (2009). Direct Compute: Bring GPU computing to the mainstream. In *GPU technology conference* (p. 23).

Nimmagadda, Y., Kumar, K., Lu, Y.-H. & Lee, C. G. (2010). Real-time moving object recognition and tracking using computation offloading. In *Intelligent robots and systems (IROS), 2010 IEEE/RSJ*

international conference on (pp. 2449–2455). doi: 10.1109/IROS.2010.5650303

Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J. & Walker, K. R. (1997). Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on operating systems principles* (pp. 276–287). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/268998.266708 doi: 10.1145/268998.266708

NVIDIA Corporation. (2016a). *CUDA best practices guide.* Retrieved 2016-11-09, from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

NVIDIA Corporation. (2016b). *CUDA parallel programming and computing platform.* Retrieved 2016-11-09, from http://www.nvidia.com/object/cuda_home_new.html

NVIDIA Corporation. (2016c). *Thrust parallel algorithms library.* Retrieved 2016-11-09, from https://developer.nvidia.com/thrust

O'Hara, K. J., Nathuji, R., Raj, H., Schwan, K. & Balch, T. (2006). AutoPower: toward energy-aware software systems for distributed mobile robots. In *Proceedings 2006 IEEE international conference on robotics and automation, 2006. ICRA 2006.* (p. 2757-2762). doi: 10.1109/ROBOT.2006.1642118

Ok, M., Seo, J.-W. & Park, M.-s. (2007). A distributed resource furnishing to offload resource-constrained devices in cyber foraging toward pervasive computing. In T. Enokido, L. Barolli & M. Takizawa (Eds.), *Network-based information systems: First international conference, NBiS 2007, Regensburg, Germany, September 3-7, 2007. proceedings* (pp. 416–425). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-540-74573-0_43 doi: 10.1007/978-3-540-74573-0_43

Olson, C. F. (1995). Parallel algorithms for hierarchical clustering. *Parallel Computing*, *21*(8), 1313-1325. Retrieved from http://www.sciencedirect.com/science/article/pii/016781919500017I doi: http://dx.doi.org/10.1016/0167-8191(95)00017-I

O'Sullivan, M. J. & Grigoras, D. (2013). The cloud personal assistant for providing services to mobile clients. In *Service oriented system engineering (SOSE), 2013 IEEE 7th international symposium on* (p. 478-485). doi: 10.1109/SOSE.2013.39

Ou, S., Wu, Y., Yang, K. & Zhou, B. (2008). Performance analysis of fault-tolerant offloading systems for pervasive services in mobile wireless environments. In *ICC'08. IEEE international conference on communications* (pp. 1856–1860). doi: 10.1109/ICC.2008.356

Ou, S., Yang, K. & Liotta, A. (2006). An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In *Fourth annual IEEE international conference on pervasive computing and communications (PERCOM'06)* (p. 10 pp.-125). doi: 10.1109/PERCOM.2006.7

Ou, S., Yang, K. & Zhang, J. (2007). An effective offloading middleware for pervasive services on mobile devices. *Pervasive and Mobile Computing*, *3*(4), 362 - 385. Retrieved from http://www.sciencedirect.com/science/article/pii/S1574119207000338 (Middleware for Pervasive Computing) doi: http://dx.doi.org/10.1016/j.pmcj.2007.04.004

Paniagua, C., Flores, H. & Srirama, S. N. (2012). Mobile sensor data classification for human activity recognition using mapreduce on cloud. *Procedia Computer Science*, *10*, 585 - 592. Retrieved from http://www.sciencedirect.com/science/article/pii/S1877050912004322 doi: http://

dx.doi.org/10.1016/j.procs.2012.06.075

Park, C. Y. (1993). Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, *5*(1), 31–62. Retrieved from http://dx.doi.org/10.1007/BF01088696 doi: 10.1007/BF01088696

Păsăreanu, C. S. & Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, *11*(4), 339–353. Retrieved from http://dx.doi.org/10.1007/s10009-009-0118-1 doi: 10.1007/s10009-009-0118-1

Paterson, M., Lindsay, D., Monotti, A. & Chin, A. (2007). Dart: A new missile in australia's e-research strategy. *Online Information Review*, *31*(2), 116-134. doi: 10.1108/14684520710747185

Pathak, A., Hu, Y. C., Zhang, M., Bahl, P. & Wang, Y.-M. (2011). Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on computer systems* (pp. 153–168). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1966445.1966460 doi: 10.1145/1966445.1966460

Peffers, K., Rothenberger, M., Tuunanen, T. & Vaezi, R. (2012). Design science research evaluation. In *Proceedings of the 7th international conference on design science research in information systems: Advances in theory and practice* (pp. 398–410). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-29863-9_29 doi: 10.1007/978-3-642-29863-9_29

Peffers, K., Tuunanen, T., Rothenberger, M. & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77. Retrieved from http://dx.doi.org/10.2753/MIS0742-1222240302 doi: 10.2753/MIS0742-1222240302

Petrini, F., Kerbyson, D. J. & Pakin, S. (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on supercomputing* (pp. 55–). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1048935.1050204 doi: 10.1145/1048935.1050204

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (1996). *Numerical recipes in C* (Vol. 2). Cambridge university press Cambridge.

Pries-Heje, J., Baskerville, R. & Venable, J. (2008). Strategies for design science research evaluation. In *Proceedings of the 16th European conference on information systems.* Retrieved from http://aisel.aisnet.org/ecis2008/87/

Puschner, P. P. & Schedl, A. V. (1997). Computing maximum task execution times — a graph-based approach. *Real-Time Systems*, *13*(1), 67–91. Retrieved from http://dx.doi.org/10.1023/A:1007905003094 doi: 10.1023/A:1007905003094

R Core Team. (2016). *R source code: hclust.* Retrieved 2016-11-09, from https://svn.r-project.org/R/trunk/src/library/stats/R/hclust.R

Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D. & Govindan, R. (2011). Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on mobile systems, applications, and services* (pp. 43–56). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1999995.2000000 doi: 10.1145/1999995.2000000

Rachuri, K. K., Mascolo, C., Musolesi, M. & Rentfrow, P. J. (2011). SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th annual international conference on mobile computing and networking* (pp. 73–84). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2030613.2030623 doi: 10.1145/2030613.2030623

Rahimi, M. R., Venkatasubramanian, N., Mehrotra, S. & Vasilakos, A. V. (2012). MAPCloud: Mobile applications on an elastic and scalable 2-tier cloud architecture. In *Proceedings of the 2012 IEEE/ACM fifth international conference on utility and cloud computing* (pp. 83–90). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/UCC.2012.25 doi: 10.1109/UCC.2012.25

Rajasekaran, S. (2005). Efficient parallel hierarchical clustering algorithms. *IEEE Transactions on Parallel Distributed Systems*, *16*(6), 497–502. Retrieved from http://dx.doi.org/10.1109/TPDS.2005.72 doi: 10.1109/TPDS.2005.72

Rasmussen, E. M. (1992). Clustering algorithms. *Information retrieval: data structures & algorithms*, *419*, 442.

Rawassizadeh, R., Anjomshoaa, A. & Tomitsch, M. (2011). A framework for long-term archiving of pervasive device information. In *Proceedings of the 9th international conference on advances in mobile computing and multimedia* (pp. 244–247). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2095697.2095747 doi: 10.1145/2095697.2095747

Rego, P. A., Costa, P. B., Coutinho, E. F., Rocha, L. S., Trinta, F. A. & de Souza, J. N. (2016). Performing computation offloading on multiple platforms. *Computer Communications*. Retrieved from http://www.sciencedirect.com/science/article/pii/S0140366416302869 doi: http://dx.doi.org/10.1016/j.comcom.2016.07.017

Reistad, B. & Gifford, D. K. (1994). Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM conference on LISP and functional programming* (pp. 65–78). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/182409.182439 doi: 10.1145/182409.182439

Rigole, P., Berbers, Y. & Holvoet, T. (2004). Mobile adaptive tasks guided by resource contracts. In *Proceedings of the 2nd workshop on middleware for pervasive and ad-hoc computing* (pp. 117–120). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1028509.1028512 doi: 10.1145/1028509.1028512

Rim, H., Kim, S., Kim, Y. & Han, H. (2006). Transparent method offloading for slim execution. In *2006 1st international symposium on wireless pervasive computing* (p. 1-6). doi: 10.1109/ISWPC.2006.1613608

Robinson, D. & Foulds, L. (1981). Comparison of phylogenetic trees. *Mathematical Biosciences*, *53*(1), 131 - 147. Retrieved from http://www.sciencedirect.com/science/article/pii/0025556481900432 doi: http://dx.doi.org/10.1016/0025-5564(81)90043-2

Rong, P. & Pedram, M. (2003). Extending the lifetime of a network of battery-powered mobile devices by remote processing: A markovian decision-based approach. In *Proceedings of the 40th annual*

design automation conference (pp. 906–911). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/775832.776060 doi: 10.1145/775832.776060

Saab, S. A., Saab, F., Kayssi, A., Chehab, A. & Elhajj, I. H. (2015). Partial mobile application offloading to the cloud for energy-efficiency with security measures. *Sustainable Computing: Informatics and Systems*, *8*, 38 - 46. Retrieved from http://www.sciencedirect.com/science/article/pii/S2210537915000396 (Special Issue on Computing for a Greener Water/Energy/Emissions Nexus; edited by Carol J. Miller.andSpecial Issue on Green Mobile Cloud Computing (Green MCC); edited by Danielo G. Gomes, Rafael Tolosana-Calasanz, and Nazim Agoulmine.) doi: http://dx.doi.org/10.1016/j.suscom.2015.09.002

Sander, J., Qin, X., Lu, Z., Niu, N. & Kovarsky, A. (2003). Automatic extraction of clusters from hierarchical clustering representations. In *Proceedings of the 7th Pacific-Asia conference on advances in knowledge discovery and data mining* (pp. 75–87). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1760894.1760906

Sarkar, V. (1989). Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN 1989 conference on programming language design and implementation* (pp. 298–312). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/73141.74845 doi: 10.1145/73141.74845

Satyanarayanan, M., Bahl, P., Caceres, R. & Davies, N. (2009). The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, *8*(4), 14–23. Retrieved from http://dx.doi.org/10.1109/MPRV.2009.82 doi: 10.1109/MPRV.2009.82

Scharff, C. & Verma, R. (2010). Scrum to support mobile application development projects in a just-in-time learning context. In *Proceedings of the 2010 ICSE workshop on cooperative and human aspects of software engineering* (pp. 25–31). New York, NY, USA: ACM.

Scornavacca, C., Zickmann, F. & Huson, D. H. (2011). Tanglegrams for rooted phylogenetic trees and networks. *Bioinformatics*, *27*(13), i248-i256. Retrieved from http://bioinformatics.oxfordjournals.org/content/27/13/i248.abstract doi: 10.1093/bioinformatics/btr210

Sedita, S. R. (2012). Leveraging the intangible cultural heritage: Novelty and innovation through exaptation. *City, Culture and Society*, *3*(4), 251 - 259. Retrieved from http://www.sciencedirect.com/science/article/pii/S1877916612000744 doi: http://dx.doi.org/10.1016/j.ccs.2012.11.009

Seelam, S., Fong, L., Tantawi, A., Lewars, J., Divirgilio, J. & Gildea, K. (2013). Extreme scale computing: Modeling the impact of system noise in multi-core clustered systems. *Journal of Parallel and Distributed Computing*, *73*(7), 898 - 910. Retrieved from http://www.sciencedirect.com/science/article/pii/S0743731513000178 (Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012) doi: http://dx.doi.org/10.1016/j.jpdc.2013.01.016

Seshasayee, B., Nathuji, R. & Schwan, K. (2007). Energy-aware mobile service overlays: Cooperative dynamic power management in distributed mobile systems. In *Fourth international conference on autonomic computing (ICAC'07)* (p. 6-6). doi: 10.1109/ICAC.2007.14

Seshia, S. A. & Rakhlin, A. (2008). Game-theoretic timing analysis. In *Proceedings of the 2008 IEEE/ACM*

international conference on computer-aided design (pp. 575–582). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=1509456.1509584

Shalom, S. A. & Dash, M. (2013). Efficient partitioning based hierarchical agglomerative clustering using graphics accelerators with CUDA. *International Journal of Artificial Intelligence & Applications*, *4*(2), 13.

Shalom, S. A. A., Dash, M. & Tue, M. (2010). An approach for fast hierarchical agglomerative clustering using graphics processors with CUDA. In M. J. Zaki, J. X. Yu, B. Ravindran & V. Pudi (Eds.), *Advances in knowledge discovery and data mining: 14th Pacific-Asia conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010. proceedings. part II* (pp. 35–42). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-13672-6_4 doi: 10.1007/978-3-642-13672-6_4

Shalom, S. A. A., Dash, M., Tue, M. & Wilson, N. (2009). Hierarchical agglomerative clustering using graphics processor with compute unified device architecture. In *2009 international conference on signal processing systems* (p. 556-561). doi: 10.1109/ICSPS.2009.167

Shi, C., Habak, K., Pandurangan, P., Ammar, M., Naik, M. & Zegura, E. (2014). COSMOS: Computation offloading as a service for mobile devices. In *Proceedings of the 15th ACM international symposium on mobile ad hoc networking and computing* (pp. 287–296). New York, NY, USA: ACM. Retrieved from http://doi.acm.org.elibrary.jcu.edu.au/10.1145/2632951.2632958 doi: 10.1145/2632951.2632958

Shi, C., Lakafosis, V., Ammar, M. H. & Zegura, E. W. (2012). Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth ACM international symposium on mobile ad hoc networking and computing* (pp. 145–154). New York, NY, USA: ACM. Retrieved from http://doi.acm.org.elibrary.jcu.edu.au/10.1145/2248371.2248394 doi: 10.1145/2248371.2248394

Shi, C., Pandurangan, P., Ni, K., Yang, J., Ammar, M., Naik, M. & Zegura, E. (2013). IC-Cloud: Computation offloading to an intermittently-connected cloud.

Shiraz, M., Gani, A., Shamim, A., Khan, S. & Ahmad, R. W. (2015). Energy efficient computational offloading framework for mobile cloud computing. *Journal of Grid Computing*, *13*(1), 1–18. Retrieved from http://dx.doi.org/10.1007/s10723-014-9323-6 doi: 10.1007/s10723-014-9323-6

Shiraz, M., Sookhak, M., Gani, A. & Shah, S. A. A. (2015). A study on the critical analysis of computational offloading frameworks for mobile cloud computing. *Journal of Network and Computer Applications*, *47*, 47 - 60. Retrieved from http://www.sciencedirect.com/science/article/pii/S1084804514002100 doi: http://dx.doi.org/10.1016/j.jnca.2014.08.011

Sibson, R. (1973). Slink: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, *16*(1), 30-34. Retrieved from http://comjnl.oxfordjournals.org/content/16/1/30.abstract doi: 10.1093/comjnl/16.1.30

Simanta, S., Ha, K., Lewis, G., Morris, E. & Satyanarayanan, M. (2013). A reference architecture for mobile code offload in hostile environments. In D. Uhler, K. Mehta & J. L. Wong (Eds.), *Mobile computing, applications, and services: 4th international conference, MobiCASE 2012, Seattle, WA,*

USA, October 11-12, 2012. Revised selected papers (pp. 274–293). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-36632-1_16 doi: 10.1007/978-3-642-36632-1_16

Sneath, P. H. A. (1957). The application of computers to taxonomy. *Microbiology*, *17*(1), 201-226. Retrieved from http://mic.microbiologyresearch.org/content/journal/micro/10.1099/00221287-17-1-201

Sottile, M. & Minnich, R. (2004). Analysis of microbenchmarks for performance tuning of clusters. In *2004 IEEE international conference on cluster computing* (p. 371-377). doi: 10.1109/CLUSTR.2004.1392636

Sottile, M. J., Chandu, V. P. & Bader, D. A. (2006). Performance analysis of parallel programs via message-passing graph traversal. In *Proceedings of the 20th international conference on parallel and distributed processing* (pp. 84–84). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=1898953.1899017

Srirama, S. N., Paniagua, C. & Flores, H. (2012). Social group formation with mobile cloud services. *Service Oriented Computing and Applications*, *6*(4), 351–362. Retrieved from http://dx.doi.org/10.1007/s11761-012-0111-5 doi: 10.1007/s11761-012-0111-5

Staelin, C. (2005). lmbench: an extensible micro-benchmark suite. *Software: Practice and Experience*, *35*(11), 1079–1105. Retrieved from http://dx.doi.org/10.1002/spe.665 doi: 10.1002/spe.665

Staelin, C. & McVoy, L. (1998). Mhz: Anatomy of a micro-benchmark. In *Proceedings of the annual conference on USENIX annual technical conference* (pp. 13–13). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1268256.1268269

Stappert, F. & Altenbernd, P. (2000). Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, *46*(4), 339 - 355. Retrieved from http://www.sciencedirect.com/science/article/pii/S1383762199000107 doi: http://dx.doi.org/10.1016/S1383-7621(99)00010-7

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 111–147.

Su, Y.-Y. & Flinn, J. (2005). Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd international conference on mobile systems, applications, and services* (pp. 79–92). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1067170.1067180 doi: 10.1145/1067170.1067180

Tantono, M. (2015). Parallelisation of hierarchical clustering algorithms for metagenomics.

Theiling, H. (2002). Control flow graphs for real-time systems analysis. *Universität des Saarlandes, Diss*.

Tolia, N., Andersen, D. G. & Satyanarayanan, M. (2006). Quantifying interactive user experience on thin clients. *Computer*, *39*(3), 46–52. Retrieved from http://dx.doi.org/10.1109/MC.2006.101 doi: 10.1109/MC.2006.101

Tsafrir, D. (2007). The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Proceedings of the 2007 workshop on experimental computer science.* New

York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1281700.1281704 doi: 10.1145/1281700.1281704

Tsafrir, D., Etsion, Y., Feitelson, D. G. & Kirkpatrick, S. (2005). System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on supercomputing* (pp. 303–312). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1088149.1088190 doi: 10.1145/1088149.1088190

Tukey, J. W. (1977). *Exploratory data analysis*. Boston, MA, USA: Addison-Wesley.

Venable, J., Pries-Heje, J. & Baskerville, R. (2012). A comprehensive framework for evaluation in design science research. In *Proceedings of the 7th international conference on design science research in information systems: Advances in theory and practice* (pp. 423–438). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-29863-9_31 doi: 10.1007/978-3-642-29863-9_31

Verbelen, T., Hens, R., Stevens, T., De Turck, F. & Dhoedt, B. (2010). Adaptive online deployment for resource constrained mobile smart clients. In Y. Cai, T. Magedanz, M. Li, J. Xia & C. Giannelli (Eds.), *Mobile wireless middleware, operating systems, and applications: Third international conference, Mobilware 2010, Chicago, IL, USA, June 30 - July 2, 2010. Revised selected papers* (pp. 115–128). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-17758-3_9 doi: 10.1007/978-3-642-17758-3_9

Verbelen, T., Simoens, P., Turck, F. D. & Dhoedt, B. (2012). AIOLOS: Middleware for improving mobile application performance through cyber foraging. *Journal of Systems and Software*, *85*(11), 2629 - 2639. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121212001641 doi: http://dx.doi.org/10.1016/j.jss.2012.06.011

Verbelen, T., Stevens, T., Simoens, P., Turck, F. D. & Dhoedt, B. (2011). Dynamic deployment and quality adaptation for mobile augmented reality applications. *Journal of Systems and Software*, *84*(11), 1871 - 1882. Retrieved from http://www.sciencedirect.com/science/article/pii/S016412121100166X (Mobile Applications: Status and Trends) doi: http://dx.doi.org/10.1016/j.jss.2011.06.063

Walisch, K. (2014). *primesieve - fast prime number generator.* Retrieved 2014-07-07, from http://primesieve.org/

Wang, B., Ding, Q. & Rahal, I. (2008). Parallel hierarchical clustering on market basket data using weighted confidence affinity. In *2008 IEEE international conference on data mining workshops* (pp. 526–532).

Wang, B. & Dong, A. (2008). A parallel attractor-tree based clustering method. In *HPCNCS* (pp. 176–180).

Wang, C. & Li, Z. (2004). Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 conference on programming language design and implementation* (pp. 119–130). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/996841.996857 doi: 10.1145/996841.996857

Wang, S., Kodase, S., Shin, K. G. & Kiskis, D. L. (2002). Measurement of OS services and its application to performance modeling and analysis of integrated embedded software. In *Proceedings of the eighth IEEE real-time and embedded technology and applications symposium (RTAS'02)* (pp. 113–).

Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=827265.828508

Weinsberg, Y., Dolev, D., Wyckoff, P. & Anker, T. (2007). Accelerating distributed computing applications using a network offloading framework. In *Parallel and distributed processing symposium, 2007. IPDPS 2007. IEEE international* (pp. 1–10). doi: 10.1109/IPDPS.2007.370319

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., ... Stenström, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, *7*(3), 36:1–36:53. Retrieved from http://doi.acm.org/10.1145/1347375.1347389 doi: 10.1145/1347375.1347389

Wilson, J., Dai, M., Jakupovic, E., Watson, S. & Meng, F. (2007). Supercomputing with toys: harnessing the power of NVIDIA 8800GTX and playstation 3 for bioinformatics problems. In *Computational systems bioinformatics conference* (Vol. 6, pp. 387–390).

Wolfinger, R. (2008). Plug-in architecture and design guidelines for customizable enterprise applications. In *Companion to the 23rd ACM SIGPLAN conference on object-oriented programming systems languages and applications* (pp. 893–894). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1449814.1449895 doi: 10.1145/1449814.1449895

Wolski, R., Gurun, S., Krintz, C. & Nurmi, D. (2008). Using bandwidth data to make computation offloading decisions. In *IPDPS 2008. IEEE international symposium on parallel and distributed processing* (pp. 1–8). doi: 10.1109/IPDPS.2008.4536215

Wu, C.-H., Horng, S.-J. & Tsai, H.-R. (2000). Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical buses. *Journal of Parallel and Distributed Computing*, *60*(9), 1137 - 1153. Retrieved from http://www.sciencedirect.com/science/article/pii/S0743731500916447 doi: http://dx.doi.org/10.1006/jpdc.2000.1644

Xian, C., Lu, Y.-H. & Li, Z. (2007). Adaptive computation offloading for energy conservation on battery-powered systems. In *Proceedings of the 13th international conference on parallel and distributed systems - Volume 01* (pp. 1–8). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dx.doi.org/10.1109/ICPADS.2007.4447724 doi: 10.1109/ICPADS.2007.4447724

Yang, K., Ou, S. & Chen, H. H. (2008). On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *IEEE Communications Magazine*, *46*(1), 56-63. doi: 10.1109/MCOM.2008.4427231

Yang, L., Cao, J., Yuan, Y., Li, T., Han, A. & Chan, A. (2013). A framework for partitioning and execution of data stream applications in mobile cloud computing. *SIGMETRICS Performance Evaluation Review*, *40*(4), 23–32. Retrieved from http://doi.acm.org/10.1145/2479942.2479946 doi: 10.1145/2479942.2479946

Yousafzai, A., Gani, A., Noor, R. M., Naveed, A., Ahmad, R. W. & Chang, V. (2016). Computational offloading mechanism for native and android runtime based mobile applications. *Journal of Systems and Software*, *121*, 28 - 39. Retrieved from http://www.sciencedirect.com/science/article/pii/S0164121216301364 doi: http://dx.doi.org/10.1016/j.jss.2016.07.043

Zhang, Q. & Zhang, Y. (2006). Hierarchical clustering of gene expression profiles with graphics

hardware acceleration. *Pattern Recognition Letters*, *27*(6), 676-681. Retrieved from http://www.sciencedirect.com/science/article/pii/S0167865505002965 doi: http://dx.doi.org/10.1016/j.patrec.2005.06.016

Zhang, T., Ramakrishnan, R. & Livny, M. (1996). BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD international conference on management of data* (pp. 103–114). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/233269.233324 doi: 10.1145/233269.233324

Zhang, X., Jeon, W., Gibbs, S. & Kunjithapatham, A. (2012). Elastic HTML5: Workload offloading using cloud-based web workers and storages for mobile devices. In M. Gris & G. Yang (Eds.), *Mobile computing, applications, and services: Second international ICST conference, MobiCASE 2010, Santa Clara, CA, USA, October 25-28, 2010, revised selected papers* (pp. 373–381). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/978-3-642-29336-8_26 doi: 10.1007/978-3-642-29336-8_26

Zhang, X., Kunjithapatham, A., Jeong, S. & Gibbs, S. (2011). Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing. *Mobile Networks and Applications*, *16*(3), 270–284. Retrieved from http://dx.doi.org/10.1007/s11036-011-0305-7 doi: 10.1007/s11036-011-0305-7

Zhang, Y., Huang, G., Liu, X., Zhang, W., Mei, H. & Yang, S. (2012). Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications* (pp. 233–248). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2384616.2384634 doi: 10.1145/2384616.2384634

Zhang, Y., t. Guan, X., Huang, T. & Cheng, X. (2009). A heterogeneous auto-offloading framework based on web browser for resource-constrained devices. In *Internet and web applications and services, 2009. ICIW '09. fourth international conference on* (p. 193-199). doi: 10.1109/ICIW.2009.35