Reactive Execution-Time Forecasting of Dynamically-Adaptable Software

Shane Brennan

A Dissertation submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

May 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Shane Brennan

Dated: May 29, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this Dissertation upon request.

Shane Brennan

Dated: May 29, 2011

Acknowledgements

This thesis, I hope, will stand as some meagre monument to the unwavering support of my parents, who place their hope and trust in education as a means of improving the lives of their children. It is dedicated to the late Mrs. Catherine Grace, who quite literally, helped me on my first steps along the road to school. Lastly, it is dedicated to my grandparents, who were committed to the belief that a better life could be won by dedication and learning.

First, and foremost, I would like to thank my supervisor Dr. Siobhán Clarke for her tireless work, counsel and guidance, and without whom the completion of this work would have been impossible. I would also like to thank Dr. Vinny Cahill, Dr. Bill Harrison, Dr. René Meier, Dr. Myra O'Regan and all the staff in the Distributed Systems Group for their help during a thoroughly enjoyable, but definitely taxing, few years in Trinity College Dublin. Finally, I would like to thank all the players, members and supporters of Bohemian Football Club for constantly reminding me that football is not a simple matter of life and death, but something much, much more important.

Shane Brennan

University of Dublin, Trinity College

May 2011

Abstract

Software operating in domains such as process management systems, wireless sensor networks and spacecraft control systems are expected to continue uninterrupted operation over extended periods, without any manual supervision, maintenance or external intervention. However, unexpected events or changes in the operating environment over time, require the software to occasionally update itself to ensure correct operation over a prolonged interval. These updates to software behaviour may be achieved by a process known as dynamic software adaptation.

Adapting software dynamically allows it to respond to unexpected operational challenges, to update unwanted or unnecessary functionality, and to optimize its behaviour to fit the prevailing operating conditions. However, adaptations can also unintentionally alter the execution time of the software. In this way, timing delays, missed deadlines and functional errors may be unwittingly introduced into an otherwise dependable codebase. Estimating the likely execution time of dynamically-adaptable software is critical to avoid functional interference caused by timing uncertainty. Unfortunately, predicting the execution time of dynamically-adaptable software cannot be accomplished using traditional timing analysis methods, without halting the system or restricting the set of adaptable software behaviours. Static timing analysis methods cannot re-evaluate timing estimates at runtime, since they require a lengthy off-line analysis period. Conversely, measurement-based dynamic timing analysis methods cannot provide any timing estimates immediately following an adaptation, until a large number of observations have been recorded and evaluated.

Reliably and precisely estimating the execution time provides assurances about the suitability of the dynamically-adaptable software within its current operating environment, as well as indicating the likely improvement in timing behaviour due to recent functional adaptations. The research question addressed by this thesis is whether adaptive statistical methods, applied at

runtime, can accurately predict the timing behaviour of dynamically-adaptable software.

To address this question, this thesis describes the application of statistical methods at run-time to predict the timing behaviour of dynamically-adaptable software. Using a dynamically-generated predictive model, forecasts are made about the likely execution time of the current configuration of the software, as well as allowing estimates to be generated describing the probabilistic timing impact of functional adaptations.

The contributions of this thesis are three-fold. Firstly, the timing behaviour of a dynamically-adaptable software system can be accurately and precisely predicted at runtime using statistical methods. Next, these predictions can be generated with limited prior warning and without halting the system to perform the analysis, restricting the scope of adaptations or relying on extensive off-line generated measurements. Lastly, timing predictions for dynamically adaptable software can be used as feedback into the adaptation process itself, to select the most appropriate configuration of the software for the prevailing operating conditions. A dynamically-adaptable software system, executing on a resource-constrained embedded device, is used to evaluate this predictive model. The timing estimates produced at run-time show that the accuracy and precision are only slightly below what would be achieved using a well-established static timing analysis method executed offline under ideal circumstances.

Contents

Acknowledgements **iv**

Abstract **iv**

List of Tables **xi**

List of Figures **xii**

Chapter 1 Introduction **1**

1.1 Dynamically-Adaptable Software 2

1.1.1 Component-Based Software 5

1.2 Software Timeliness 8

1.2.1 Execution-Time Requirements 9

1.2.2 Predicting Software Execution Times 11

1.2.3 Challenges 13

1.3 Contribution 14

1.4 Assumptions 15

1.5 Issues Not Covered 18

1.6 Evaluation 19

1.7 Road Map 19

1.8 Summary 20

Chapter 2 Related Work **21**

2.1 Software Adaptation 23

2.1.1 Static Composition 26

2.1.2 Compile-time Adaptation 30
2.1.3 Load/Link-time Adaptation 32
2.1.4 Dynamic Adaptation 34
2.1.5 Summary of Adaptation Frameworks 38
2.2 Timing Analysis 40
2.2.1 Static Timing Analysis 43
2.2.1.1 Static Analysis Frameworks 43
2.2.1.2 Formal Methods Analysis 45
2.2.1.3 System Models 47
2.2.1.4 Tool-Based Timing Analysis 50
2.2.2 Dynamic Timing Analysis 52
2.2.2.1 Measurement-driven Analysis 53
2.2.2.2 Statistical Analysis 55
2.3 Chapter Summary 57

Chapter 3 TimePredict: A Reactive Run-time Timing Analysis 60

3.1 Software Timeliness 61
3.1.1 Operational restrictions 66
3.2 The TimePredict Approach 68
3.2.1 Timing Measurement 71
3.2.1.1 Clock Resolution 73
3.2.1.2 Clock Drift 74
3.2.2 Data Selection 75
3.3 Average-Case Analysis 77
3.3.1 Exponential Smoothing Model 78
3.3.1.1 Critique of the ES Model 82
3.4 Worst-Case Analysis 83
3.4.1 Initial Worst-Case Heuristic 84
3.4.2 Generalized Extreme Value Model 87
3.4.2.1 Gumbel Distribution 87
3.4.2.2 Fréchet Distribution 88
3.4.2.3 Weibull Distribution 90

3.4.2.4 Generalised Pareto Distribution 91
3.4.3 Model Fitting 93
3.4.3.1 Goodness of Fit 94
3.4.3.2 Generating Estimates 96
3.5 Validation and Feedback 96

Chapter 4 TimePredict Implementation 100

4.1 Operational Implementation 101
4.1.1 Target Operating Environment 104
4.1.2 Hardware Considerations 104
4.1.2.1 Wireless Communication 106
4.1.2.2 Sun SPOT sensors 109
4.1.3 Timing Considerations 110
4.1.4 Resource Usage Considerations 111
4.1.4.1 Memory Overhead 111
4.1.4.2 Processing Overhead 113
4.1.5 Accuracy and Clock Resolution 114
4.1.6 Estimate Update Frequency 116
4.1.7 Impact on the System 116
4.2 Software Implementation 117
4.2.1 Timing Measurement 118
4.2.1.1 Adding Timing Measurements 119
4.2.2 Algorithm Optimizations 122
4.2.3 Model Validation 124
4.3 Dynamically-Adaptable Application Scenario 130
4.3.1 Experimental Setup 131
4.3.2 Experimental Goals 133
4.3.3 Application Configurations 134
4.3.3.1 Normal Operation 136
4.4 Summary 140

Chapter 5 Evaluation **141**

5.1 Off-line execution using benchmark measurements. 144
5.2 Analysis Setup 146
5.2.1 Statistical Analysis Tools 147
5.2.1.1 Independent Two-Sample T-Test 149
5.2.1.2 Non-Parametric Tests 149
5.2.1.3 Correlation Tests 150
5.3 TimePredict Evaluation 150
5.3.1 Software Timeliness 151
5.3.2 TimePredict Performance 154
5.3.3 System Impact 159
5.3.3.1 Timing Overhead 159
5.3.3.2 Memory Consumption 163
5.3.3.3 Power Consumption 166
5.4 Off-line Analysis 168
5.4.1 Comparative Off-line Statistical Analysis 168
5.4.2 Effect of Model Parameters on Predictive Performance 171
5.4.3 Timing Feedback 172
5.5 Summary 173

Chapter 6 Conclusion **174**

6.1 Contribution 174
6.2 Future Work 177
6.3 Conclusion 178

Appendix A Software Execution Times **179**

Bibliography **186**

List of Tables

2.1 Summary of adaptation approaches. 39
2.2 Summary of timing analysis approaches. 58

3.1 Effect of measurement availability on timing estimates. 65
3.2 Confidence intervals and critical values for the Kolmogorov-Smirnov two-sample test. 95

4.1 Available clock speeds and maximum durations on the Java Sun SPOT mote. . . 115
4.2 Execution times of the software benchmark suite, over 20 configurations. 128
4.3 Configurations of the sensor software. 135

5.1 Summary of the expected contribution matched to particular evaluations. 143
5.2 Off-line analysis performed by TimePredict on benchmark measurements. 144
5.3 Evaluation of the execution time of the sensor analysis function. 151
5.4 Correlations of configuration setup with execution time analysis. 152
5.5 TimePredict forecasting accuracy and precision. 155
5.6 Correlation between accuracy/precision and the overall range and IQR. 158
5.7 Sensor analysis timeliness with and without TimePredict. 162
5.8 Statistical tests for any timing interference due to TimePredict. 163
5.9 Memory usage with and without TimePredict. 163
5.10 95% confidence intervals for memory overhead of TimePredict. 165
5.11 Accuracy and precision of an equivalent off-line timing analysis. 169

List of Figures

1.1 Dynamic software adaptation. 3
1.2 Architecture of an idealized dynamically-adaptable component-based system. . . 6

3.1 Timing bounds used to define the execution-time performance of software. 62
3.2 Architecture of a dynamically-adaptable system featuring TimePredict. 69
3.3 Bounded timing estimates using the exponential smoothing model. 81
3.4 WCET Heuristic Performance. 86
3.5 Gumbel probability distribution. 89
3.6 Fréchet probability distribution. 90
3.7 Weibull probability distribution. 91
3.8 Pareto probability distribution. 92
3.9 Analysis of the difference between Data and Distribution CDFs. 94
3.10 WCET model fitting process. 97

4.1 TimePredict class diagram. 102
4.2 Java Sun SPOT mote. 105
4.3 Sending a Datagram-based discovery request. 107
4.4 Receiving a Datagram, using a blocking receive function. 108
4.5 Initializing and using a stream-based connection. 109
4.6 Taking sensor readings using the Squawk Java API. 110
4.7 Comparison of the System.arraycopy performance. 113
4.8 TimingListener interface. 119
4.9 Using the AT91 timing functionality. 120

4.10 Execution times for the combined software benchmark functions, over 20,000 measurements. 126
4.11 Boxplot showing the measurements taken of the software benchmark functions, across each of the 20 recorded configurations. 129
4.12 Experimental setup with Sun SPOT mote and Adaptation Server. 132
4.13 Execution order during normal operation. 138
4.14 Adaptation request and software deployment. 139

5.1 Box-plot showing the maximum and inter-quartile range of execution times. . . . 154
5.2 Histogram of the estimation accuracy of TimePredict. 156
5.3 Histogram of the estimation precision of TimePredict. 157
5.4 Execution time of the TimePredict functionality on the Sun SPOT mote. 160
5.5 Boxplot showing the execution time performance of TimePredict. 161
5.6 Boxplot showing the memory consumption of TimePredict. 164
5.7 Battery discharge while running software. 167

A.1 Detail of the ACET Timing Estimates for Configuration 16, with the red line representing execution time behaviour and the green lines representing the ES estimate bounds. 179
A.2 The same timing behaviour for Configuration 16, overlaid with worst-case estimate bounds. 180
A.3 Execution Times of Configurations 1 to 4 181
A.4 Execution Times of Configurations 5 to 8 182
A.5 Execution Times of Configurations 9 to 12 183
A.6 Execution Times of Configurations 13 to 16 184
A.7 The timing performance resulting from the execution of the benchmark functions. This dataset is used to evaluate TimePredict off-line. 185

Chapter 1

Introduction

Ab actu ad posse valet illatio.

From the past one can infer the future.

Latin Maxim

This thesis presents a run-time reactive prediction model to forecast the execution time of dynamically-adaptable software. The approach taken in this work demonstrates how highly-variable software execution times can be estimated using using a combination of extreme-value statistical modelling techniques in parallel with a self-updating Exponential Smoothing model. Although this thesis deals with run-time software adaptation within embedded devices, the contribution of this work lies not any novel adaptation selection mechanism, but in the estimation process that can be applied within this unique operating environment. Despite having limited system resources available, little prior warning of adaptations within the software, and a minimal period in which to perform the analysis, the composite predictive model described in this thesis provides a reliable, accurate and precise timing estimate of the changeable execution time of dynamically-adaptable software.

This chapter introduces the motivation behind this work, and defines the principal ideas underpinning dynamically-adaptable software. The volatile execution time of this software is outlined, and the challenges involved in predicting its likely future behaviour are presented. Lastly, this chapter describes the goals and the contribution of this work as a whole.

1.1 Dynamically-Adaptable Software

Software is routinely expected to operate in a correct, reliable and autonomous manner over prolonged periods without supervision or intervention. This expectation also extends to software running on resource-constrained devices executing within highly-variable operating environments. Where there exists a general reluctance to incur any downtime, i.e., where extended periods of uninterrupted operation are necessary, suspending execution to facilitate software updates or perform maintenance on the system is not ideal [Oreizy et al., 2008]. This prohibition on suspending execution mainly serves to restrict the software, once deployed and executing, to a fixed set of immutable functional behaviours. Unanticipated changes in the environment, or any unexpected operating conditions, must be handled using this pre-defined functionality. However, if the functionality remains fixed, and the demands being placed on the software by its operating environment begin to differ sufficiently from its capabilities, the performance of the system as a whole can degrade to the point where it becomes unfit for purpose.

To avoid the system becoming periodically unresponsive within such volatile operating environments, discrete modifications may be made during execution to optimize the available functionality to the prevailing conditions. Continuously adding hardware to compensate for poor software performance provides an unrealistic long-term solution, due to high costs, and the potential difficulties in gaining physical access to the system (e.g., as with satellites or embedded devices). Modifying the software at runtime provides the most practicable means of bridging the gap between any unanticipated operational challenges, and the apparent limitations of the system. By allowing software to adapt itself at run-time, its facility to handle extreme or unexpected operating conditions is increased, without negatively impacting on its availability.

The Oxford English dictionary defines adaptation as the "*the process of modifying a thing so as to suit new conditions*" [OED, 1989]. Within the context of software, it is then possible to characterize adaptation as the process of adding, removing or replacing functional elements within a system to provide a set of behaviours more suited to the current operating environment. Software adaptations are necessarily reactive processes, in that changes in the operating environment occur unexpectedly at run-time and require immediate corrective action in the form of functional modifications. Changes in the operating environment can take the form of unanticipated events such as device failures or software exceptions, as well as more gradual processes such as increases in system load, changing levels of software throughput and general software

obsolescence. Software that implements functional adaptations at run-time, i.e., without halting execution or requiring external intervention, may be described as being dynamically-adaptable. Buisson et al. define such dynamically-adaptable software as "*software that modifies or augments its functionality during execution in response to some observations about its operating environment*" [Buisson et al., 2005]. Both the scheduling and the functional scope of adaptations within dynamically-adaptable software are resolved only at run-time, with the principal limitation to the adaptation process being the availability of suitable alternate functional behaviours.

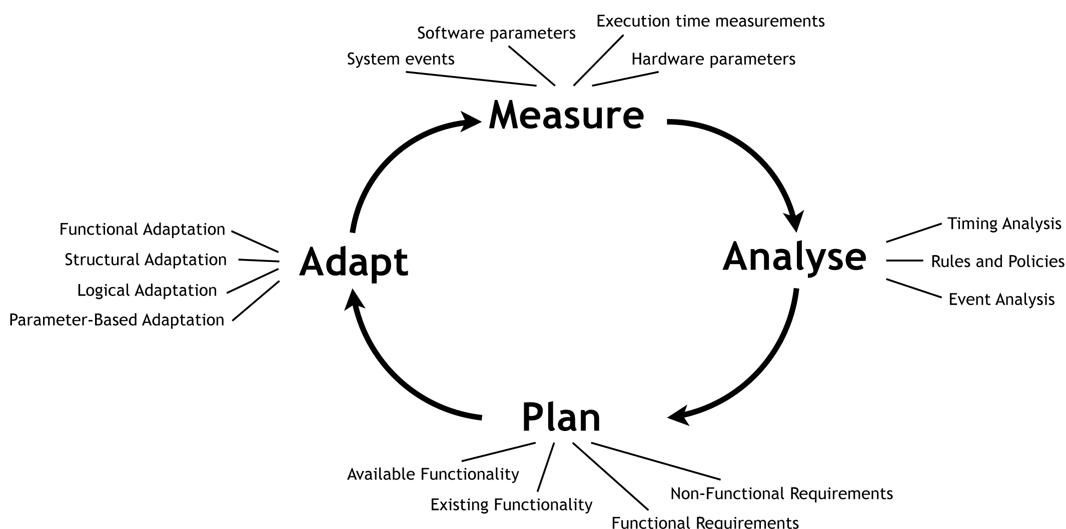


Fig. 1.1: Dynamic software adaptation.

Figure 1.1 is based upon an illustration of a control loop for self-adaptive systems, originally described in a survey paper of autonomic software systems [Dobson et al., 2006], and re-produced here in a slightly modified form. Autonomic, self-adaptive or self-* software are largely analogous to dynamically-adaptable software in intent, with the slight difference, if it exists at all, being the added emphasis on unrestricted adaptation within dynamically-adaptable software. Unrestricted adaptation permits run-time modification of the software in a manner that may not have been foreseen at design-time. In contrast, autonomic, self-adaptive and self-* software may reasonably restrict the scope of adaptations to a pre-defined set of alternate functional behaviours, in order to maintain a desired functional or non-functional property of the system [Keeney, 2004].

There are four broadly defined stages, as illustrated in Figure 1.1, that enable dynamically-adaptable software systems to respond to changes in the operating environment, namely *Measure*,

Analyse, *Plan* and *Adapt*. The first stage of an adaptation cycle involves measuring various parameters associated with the performance of the system. This provides information about the effects of the operating environment on the current configuration of the software, and highlights any performance deficits should they exist. Continuous measurement of parameters such as the system load, software interrupts and the overall software execution time allow changes in the operating environment to be inferred from changes in the measured parameters. The measurement stage facilitates an ongoing assessment of the suitability of the current configuration of the software. Next, the analysis stage evaluates the various system measurements, and determines whether firstly, an adaptation is necessary, and secondly, what are the most appropriate modifications to make to the software. Once this is complete, the planning stage takes the analysis and constructs an adaptation plan for the system. This plan is based on a determination of the available alternate software behaviours, their likely functional and non-functional performance, and the expected difference between these and the performance of the current configuration of the software. Lastly, the adaptation stage takes the completed adaptation plan, and implements the desired modifications on the executing software, either pausing execution to effect the adaptation or hot-swapping functional elements at runtime. The timing analysis approach presented in this thesis, fulfills the analysis role described in Figure 1.1, as well as enabling the measurement of software timing behaviour. Although the actual implementation of the adaptation process is largely outside the scope of this work, it is envisaged that the underlying dynamically-adaptable system uses the timing analysis to optimize its execution-time performance by dynamically adding, replacing or removing functional elements at run-time.

Some software adaptation techniques, e.g., parameter-based adaptation [Sharma et al., 2004], do not permit any new functional behaviours to be added to the software once the system is deployed. A limited form of adaptation is achieved by altering some existing parameters that govern the behaviour of the currently deployed software. In contrast, dynamically-adaptable software takes a less restrictive approach towards the adaptation process, allowing new functionality to be added much later to the system, at a time dictated by the prevailing operating conditions [Fritsch and Clarke, 2008]. A number of existing software adaptation techniques support an unrestricted approach towards run-time functional adaptation, ranging from middleware-based adaptation schemes [Sharma et al., 2004], to dynamic aspect-weaving frameworks [Assaf and Noyé, 2008], self-organising architectures [Georgiadis et al., 2002] and component-based adaptation frame-

works [Zhang et al., 2005].

1.1.1 Component-Based Software

Component-based systems provide the underlying platform for the dynamic software adaptation considered in this thesis. Software components provide a convenient means of encapsulating functionality within discrete, composable functional units. Menascé defines a software component as having a well-defined interface, allowing it to be employed in various applications for which it may or may not have been explicitly designed [Menascé et al., 2004]. Szyperski describes a software component as "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*" [Szyperski and Pfister, 1997]. Software components provide readily accessible units of adaptation, where adapting the software functionality is a matter of simply changing the components that comprise the system. In this manner, dynamically-adaptable software can be achieved through adding, replacing or removing software components from the system at runtime.

Figure 1.2 presents an idealized dynamically-adaptable component-based system, designed as a composite of a number of existing run-time adaptation frameworks. These frameworks, described in more detail in Chapter 2, share the common characteristic of allowing software to be modified during execution, by adding, removing or replacing discrete functional elements within the system. The primary focus of this thesis is on predicting the execution time of this type of dynamically-adaptable system, i.e., a system can add previously unenvisaged functional behaviours at any stage during its execution. Since the characteristics of the underlying software will influence the approach used to predict its execution-time behaviour, dynamically-adaptable systems considered within this thesis are assumed to have the following properties (labelled P1 to P6);

- P1: Adaptations are reactive, infrequently occurring events, that modify the functional behaviour of software in response to changes in the operating environment.
- P2: Adaptations occur unexpectedly during run-time, and are realized within a dynamically-adaptable system during its normal execution. In order to implement the adaptation, the system may pause execution briefly to enact the software modification, or alternatively may hot-swap functional behaviours without interrupting execution.

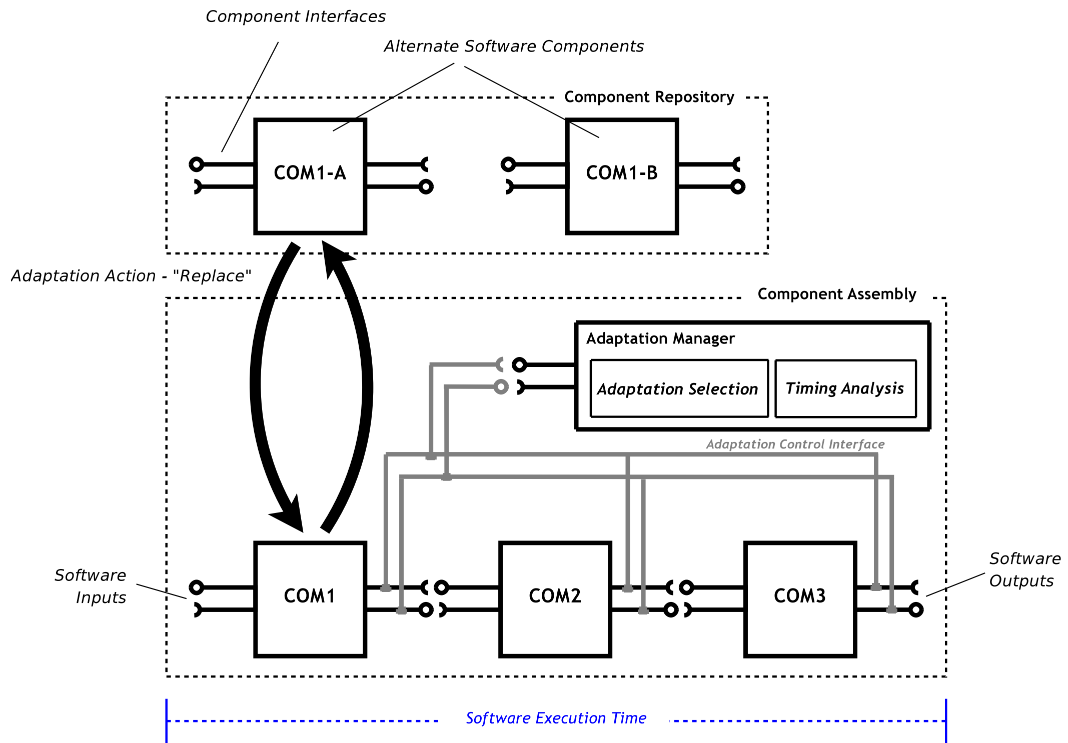


Fig. 1.2: Architecture of an idealized dynamically-adaptable component-based system.

P3: The functional scope and scheduling of an adaptation cannot be determined at design-time, and is only resolved at run-time, triggered by changes in its operating environment and the availability of alternate functionality.

P4: The adaptation process is controlled by a dedicated Adaptation Manager function within the system. The Adaptation Manager monitors, evaluates, plans and implements adaptations automatically at run-time. The goal of the Adaptation Manager is to optimize a particular property of the dynamically-adaptable system through functional adaptations, e.g., its execution time.

P5: Adaptations are achieved by adding, replacing or removing one or more software components that comprise the system. Software components are discrete units of composition, that are discoverable and composable at run-time through well-documented interfaces.

P6: Alternate software components are stored in an external component repository, and are

discoverable by the Adaptation Manager during run-time. New software components may be added to this repository at any stage during the execution of the system, and include software behaviour not originally envisaged at design-time.

A system that exhibits all of these properties may be characterized as being dynamically-adaptable. For the purposes of this thesis, the primary research question is not about the nature of dynamically-adaptable systems, nor the adaptation process in itself, but whether a statistical-based approach, implemented at run-time, can accurately and precisely estimate the timeliness of a dynamically-adaptable system. In order to illustrate an architecture for an idealized dynamically-adaptable system, and show where within such an architecture the timing analysis functionality could reside, Figure 1.2 illustrates an adaptable component-based system formed from three functional components (COM1, COM2 and COM3), as well as one control component - the adaptation manager. Both service and control interfaces are provided within each component, in the former case to interface with other components in the assembly, and in the latter to provide a mechanism for run-time supervision and intervention by the adaptation manager. The execution time of the entire component assembly is measured and recorded within the adaptation manager, where an analysis of the timeliness of the software forms an input into the adaptation selection process. An external component repository is provided with a set of alternate functional behaviours for existing components in the assembly (COM1-A and COM1-B). In the example, a 'replace' adaptation action removes the component COM1 and inserts COM1-A into the system at run-time under the direction of the adaptation manager. Software components provide a convenient level of granularity for adaptable systems, since they can leverage the often highly modularized and localized functionality within a typical codebase to decompose functional behaviours into discrete units of adaptation. Commonly, high-level functional behaviours are composed of lower-level functional entities, such as classes, objects or functions. These functional entities can be encapsulated within an individual software component, allowing each component to perform a sub-task within the overall system. Adapting the software at run-time, i.e., changing the constituent software components within the system, can be achieved using one of a number of run-time adaptation techniques such as a wrapper pattern, dynamic linking or middleware re-direction [Salehie and Tahvildari, 2009], however, within the architecture described in Figure 1.2, software components are hot-swapped into the component assembly to effect the desired adaptation.

The reactive nature of dynamically-adaptable software requires a dedicated adaptation manager to initiate, select and implement the required functional adaptations, as well as oversee the general performance of the system. A dedicated internalized adaptation manager can be encapsulated as a software component, and can execute in parallel with the software, fulfilling the various tasks illustrated in Figure 1.1. For simplicity, and to reduce the complexity of maintaining a shared system view with an external entity, it is more convenient to co-locate the adaptation manager component within the component assembly being managed. As well as managing the functional adaptations to the system, the adaptation manager includes functionality to measure and analyse the execution-time performance of the system. This software timing analysis is introduced in Section 1.2, and provides an opportune means of accessing the ongoing fitness of dynamically-adaptable systems, as well as verifying the effects on performance of run-time software adaptations.

1.2 Software Timeliness

The principal contribution of the work described in this thesis is estimating the timeliness of dynamically-adaptable software. While many other timing analysis methods currently exist [Li et al., 2007] [Marref and Bernat, 2008] [Wilhelm et al., 2008], none have been applied to evaluate the execution time of dynamically-adaptable software running within an embedded operating environment. The execution time of software may be defined as the period required to process a given task to completion, under normal operating conditions, on a specified hardware platform [Edgar, 2002]. All software, including statically-defined software operating under ideal conditions, tends to experience some minor variations in its execution time due the interaction of a large number of internal and external factors. I/O latencies, hardware interrupts, software events, available memory, networking issues, device failures, caching policies, execution history and even the ambient temperature of the processor all contribute to cause fluctuations in the timing behaviour of the system [Henia et al., 2005]. Accurately predicting the execution time allows a more efficient use to be made of the available computational resources, informs the scheduling of software tasks, and provides an on-going qualitative assessment of the performance of the software [Becker et al., 2006]. Where the software is capable of modifying its functional behaviour at run-time, its execution time can form both an input into the adaptation process, as well as a convenient means of appraising the performance of the latest configuration of the

system.

Dynamically-adaptable software responds to changes in its operating environment by changing parts of its functional make-up. However, knowing whether the operating environment has changed requires the continuous measurement of various system parameters in order to build up a 'true' picture of the current operating conditions. In effect, the status of the operating environment must be inferred from a limited set of periodic measurements. One of the more revealing properties of a software system is its ongoing execution-time performance, i.e., its timeliness. If sufficiently large or prolonged deviations from the established execution time are observed, it can be inferred that the cause for these changes is some underlying factor within the system itself or within the operating environment. For example, periods of high load on the system, such as those caused by large numbers of concurrent user requests, can overwhelm the capabilities of the system and manifest themselves as execution time delays. In this manner the timeliness of a dynamically-adaptable system can provide an indication of the capability of the software to perform its role. Since dynamically-adaptable software also permits the functionality to be modified as a reaction to changes in the operating environment (e.g., excessive delays), the timeliness of the software affords a qualitative point of comparison between candidate adaptations, and the currently deployed software configuration.

1.2.1 Execution-Time Requirements

Any predictive method that provides a reliable, accurate and precise estimate of the timeliness of dynamically-adaptable software, also provides re-assurance about the software adhering to its timing requirements. Software can be divided into three broad categories with respect to its required timing behaviour, i.e.,

- software with no explicit timing requirements
- software with soft timing requirements, and
- software with hard execution-time requirements.

The first category of timing requirements encompasses software that has been developed without any expectation of how long it will take to execute. Many software applications written today fall into this category, with the software developer eschewing execution-time considerations, focusing wholly on functional correctness [Kumar et al., 2008] . The timeliness of this type of

software forms an unconsidered, and largely ignored, by-product of its execution. In contrast, the opposite extreme is software with hard execution-time requirements, where strict inviolable deadlines are placed on the timeliness of the software. These deadlines must be adhered to over the life-time of the software, regardless of any difficulties imposed by unusual or extreme operating conditions. Typically employed within critical real-time and embedded systems, e.g., flight control systems [Mindell, 2008], hard execution-time requirements force the software to complete each task within a preset time period. To miss a timing deadline would mean the catastrophic failure of the system, since the various tasks making up the system have rigid pre-determined execution schedules. Hard real-time software is not generally amenable to run-time adaptation, or even repeated off-line modification, due to the lengthy and detailed analysis required to verify its timeliness after even minor functional modifications [Wilhelm et al., 2008]. Since the first category of software has no use for execution-time analysis, and hard real-time software effectively prohibits adaptation, the focus of this thesis is on estimating the timeliness of software with soft execution-time requirements.

Systems with soft execution-time requirements offer a balance between providing deadlines on the execution-time of the software and tolerating occasional violations of these deadlines under extreme or unusual operating conditions. Typically, the software timing deadlines are outlined by the developer in advance, or determined by the system at runtime, and aimed at ensuring software tasks execute according to pre-defined but moderately flexible timing constraints. Violations of soft timing deadlines can and do occur under extreme conditions, but only result in a reduction in the quality of service, rather than a complete failure of the system. For example, within an IP telephony application, delays in receiving packets result in a reduction in call/voice quality. Similarly, delays in rendering a frame within gaming applications result only in momentary periods of frame jitter. Unanticipated or excessive violations of soft execution-time deadlines can provide the impetus for adapting the underlying software to better suit the prevailing operating conditions. Timing deadlines offer a threshold against which the system can be measured, and if found lacking, can provide a performance target for any subsequent adaptations to the software.

Consequently, changes in the software execution time can prompt functional adaptations within dynamically-adaptable systems. However, adapting the functional behaviour of even a small part of the software can result in unintentional changes to the execution time of the software as a whole. Any functional adaptations that aim to correct or optimize performance must ensure

that these modifications do not negatively impact on software timeliness. For dynamically-adaptable software to provide any assurance of correct operation, i.e., to avoid any time-induced functional errors, it must be possible both to recognize when to adapt and to quickly predict the execution time of a newly-adapted software system.

Unfortunately, the combination of software adaptations, as well as general timing perturbations originating in the operating environment, both may contribute to unintentional alterations in the overall execution time of the software. Within closely-coupled systems with soft execution-time constraints, poorly understood timing behaviour hampers the ability of the software to function as expected. Usually, the outputs of one functional element form the inputs of the next, meaning that small localized delays can quickly propagate through the system leading to reduced throughput, missed timing deadlines, or can even precipitate functional errors in otherwise dependable code. The correct operation of a dynamically-adaptable system relies as much on reducing uncertainties about its timing behaviour as removing functional errors from the code.

1.2.2 Predicting Software Execution Times

Timing analysis is the process of evaluating software to produce an estimate of its likely execution-time within a specific operating environment. Traditionally, timing analysis has been applied to hard real-time systems, where the timeliness of the software is critical to the correct performance of the system. Specifically, the timing analysis methods applied to these critical systems are concerned solely with deriving a worst-case execution time (WCET) estimate for the software, in order to provide safety guarantees within hard real-time environments, i.e., an assurance that the software will never exceed any hard timing deadlines [Wilhelm et al., 2008]. These traditional timing analysis approaches evaluate the software timeliness using one or more static analysis methods, applied at design-time to a fixed code-base. The software persists in the same unchanged form after deployment, ensuring the timing estimate remains valid during execution.

The difference between hard real-time systems and dynamically-adaptable software is borne out in the expectations surrounding their timing behaviour. The former is expected to remain fixed once deployed and executing, while the latter is liable to change unexpectedly during execution, and continue execution with a conspicuously altered execution-time behaviour. The assumptions governing current timing analysis methods are violated when applied to dynamically-

adaptable software. The functional scope of the code is not fixed, and since execution continues immediately following an adaptation, there is no opportunity to submit the software for extended periods of off-line testing supervised by a domain expert, i.e., the programmer/developer/tester. The execution time of dynamically-adaptable software will vary considerably, and change without prior warning. As a result, software adaptations can quickly lead to a divergence between any static pre-compiled execution-time estimate and the observed execution-time behaviour.

Static timing analysis methods, i.e., those that form an off-line estimate of software timeliness, are insufficient for dynamically-adaptable software since its timing behaviour changes with every adaptation. Statically analyzing each potential configuration of a dynamically-adaptable system, similar to the approach described in the PECT framework [Hissam et al., 2003], becomes impractical when the number of potential configurations of the system grows very large. To provide an unrestricted framework for dynamic software adaptation, but yet support the prediction of its execution-time behaviour, software timing analysis must be performed concurrently with the normal operation of the system and continuously update as the timing behaviour of the system changes.

Dynamic timing analysis methods currently exist, such as those measurement-based techniques described by Petters [Petters et al., 2007] and Hansen [Hansen et al., 2009], however they require a large number of timing measurements before an estimate of the timing behaviour of the software can be created. Since adaptations change the functionality of the system, any timing measurements recorded prior to an adaptation are invalidated once the software changes. This requires a new set of timing measurements to be created before a timing estimate can be produced, potentially leading to a period immediately following an adaptation, where no timing estimates can be provided. In any dynamically adaptable system the most pressing need for an estimate of the execution time will be immediately after the system adapts, in order to establish the performance of the new configuration of the software.

Using statistical models, an accurate estimate of the timeliness of dynamically-adaptable software can be created. However, statistical models require a set of measurements to fit a selected statistical distribution to the underlying data, and then produce a valid estimate. Unfortunately, this minimum number of measurements can range from several dozen to thousands, depending on the inherent variability in the process being measured, i.e., the software timeliness. The approach outlined in this thesis, and described in more detail in Chapter 3, uses a number of

different statistical analysis techniques to provide an immediate, progressively improving estimate of the timeliness of dynamically-adaptable software.

Two aspects of the timing behaviour of the dynamically-adaptable software are analyzed, the average-case timing, and the worst-case execution time. The average-case timing estimate provides a bounded interval within which a stated percentage of the software timing measurements are expected to fall. The worst-case estimate provides a single threshold value below which another stated percentage of the timing measurements of the system are expected to occur. These two estimates provide an indication of the central tendency of the software execution-time performance and its potential for extreme behaviour, i.e., timing delays.

A linear regression analysis is applied in the first case, to supply a bounded average-case timing estimate. For the worst-case performance, a modified Holt-Winters exponential smoothing model [Chatfield, 2003] is applied initially when a small number of timing measurements are available, e.g., immediately succeeding an adaptation. The Holt-Winters model is applied to time series data, in this instance the series of software timing measurements recorded at run-time, and used to forecast the likely worst-case execution time of the dynamically adaptable software. When a sufficient number of timing measurements have been recorded, the worst-case timing estimate is generated using a statistical distribution that is fitted to the timing data at run-time. The class of statistical distribution used is known as an extreme-value distribution [Kotz and Nadarajah, 2000], and is more usually found in predicting the movement of financial markets [Poon et al., 2004] or in estimating extreme events [Alvarado et al., 1998].

These statistical modelling techniques, the collection and assimilation of timing measurements as well as the application of run-time adaptive statistical models to dynamically-adaptable software systems, are described in more detail in Chapter 3.

1.2.3 Challenges

In contrast to statically-defined software, dynamically-adaptable systems provide a flexible framework in which developers can create malleable code capable of contending with, and exploiting, highly variable operating environments. However, run-time adaptations change the timing behaviour of the software, and introduce uncertainties about its timeliness that cannot be easily analysed or predicted in the short period available during or immediately following an adaptation. To maintain a valid estimate of the execution time of dynamically-adaptable software, the

analysis process must either account for every potential configuration of the software statically (an unrealistic prospect), or perform the analysis at run-time and automatically update the timing estimate when a functional adaptation occurs. The research question central to this thesis is how statistically-based timing analysis methods can be applied to dynamically-adaptable software at run-time, without halting the system or restricting the number of potential adaptations to the software.

As the scope and scheduling of adaptations are resolved only at run-time, any estimates of software timeliness must be capable of rapidly reacting to adaptations within the system. However, since it is unlikely that any prior timing analysis will have been performed on a new configuration of the system, the timing analysis process must refresh its estimates using whatever timing information is available immediately following an adaptation. This raises a secondary research question for this work, concerning how an accurate, precise timing estimate may be produced, when the predictive model is based upon a limited set of timing measurements, recorded immediately following a software adaptation. From the sparse timing information available an estimate must then be created without interfering with the normal execution of the software, or negatively impacting on the functional behaviour of the system. Lastly, since the timing estimate is provided as feedback into the adaptation process, the predictive model used to generate the estimates must be capable of uncovering subtle changes in software timing behaviour over extended periods.

1.3 Contribution

This thesis describes the design and evaluation of an execution-time forecasting method for dynamically-adaptable software. Specifically, the approach presented in this work predicts the timeliness of software executing on a resource-constrained embedded device with soft real-time constraints. Timing predictions are generated using reactive run-time statistical methods, and are continually updated with fresh timing measurements during run-time. The contribution of this work is therefore in applying novel statistical techniques to estimate the timeliness of dynamically-adaptable software. Not only that, but to perform this analysis at run-time, within the same embedded system that is being monitored. This task is made difficult by the unique nature of the software, as well as the restrictions imposed by the operating environment.

Adaptations to the software can occur at run-time with little prior indication, and alter the

timeliness of the software in unexpected ways. The core of this thesis is a predictive model that accurately and precisely estimates software timeliness, without halting the system to perform the analysis, relying on any prior off-line testing or restricting the scope of functional adaptations. Currently, various measurement-based [Colmenares et al., 2008] [Wenzel et al., 2005], and statistical-based [Hansen et al., 2009] [Edgar, 2002] timing analysis techniques exist, however they both perform the timing analysis from an off-line context, and preclude any modifications to the underlying software at run-time. The few autonomic approaches that monitor the performance of software in real-time and adjust any QoS or performance estimates appropriately [Epifani et al., 2009] [Calinescu and Kwiatkowska, 2009], likewise do not support any timing analysis where the underlying software is dynamically adaptable. There is a gap in the current knowledge about how best to estimate the execution time of software that can both change unexpectedly during run-time, as well as modify its functionality into a configuration that may be unenvisaged at design-time.

Timing estimates are produced automatically at run-time using this predictive model, with an accuracy only slightly below what would be achieved using an industry-standard timing analysis process performed off-line in ideal circumstances and under manual supervision. In addition to providing timing estimates of the software, the outputs of this predictive model provide feedback into the adaptation process as a means of selecting the most appropriate configuration of the software for the prevailing operating conditions. Lastly, the predictive model, while interleaved with the normal execution of the software, does not negatively impact the performance of the system.

1.4 Assumptions

This thesis is grounded in a number of underlying assumptions concerning the overall approach, the performance of the execution-time forecasting process and its evaluation. In addition, there are also several fundamental assumptions about the implementation of dynamically-adaptable software, the nature of the operating environment and the limitations of the adaptation process.

Hardware: The operating environment includes both the hardware running the dynamically-adaptable software system, as well as any external factors that influence software behaviour, e.g., user requests. The hardware is presumed to be a single stand-alone device, with its own

processor(s), memory and storage, all located on the same physical unit. For the purposes of this thesis, resource-constrained embedded devices provide the target hardware environment, typically having MHz-scale processors with solid-state memory/storage not in excess of several hundred megabytes. The use of resource-constrained embedded devices is desirable since it provides a convenient motivation for application domains where both computing resources and execution time performance are critical. However, the implication of using resource-constrained devices is that the forecasting process must minimize both its memory footprint and processing overhead on the device. In addition, the limited memory cannot store a very large set of timing measurements, so the forecasting process must selectively store or discard timing data according to its freshness and/or its descriptive value, i.e., excessive execution times are more useful in calculating the likely worst-case execution time of the system, than more average measurements.

Rarely Occurring Events: The system is composed of dynamically-adaptable component-based software, and includes an internal adaptation manager with timing analysis functionality. Adaptations are assumed to be rarely occurring events, implemented at run-time by changing the composition of the component-based system, using a set of alternate software components provided within a component repository. An adaptation process that is called too frequently will impact on the normal execution of the software, since resources and time will be required to select and implement changes to the software. For a dynamically-adaptable system to optimize its functional response, minor variations within its operating environment must be overlooked, and adaptations triggered only when absolutely required. An overly reactive adaptation process may potentially create a repeating oscillation between two extreme behaviours, to the detriment of its functional goals. For a timing analysis process, estimating the execution time of a dynamically-adaptable system will require some distinct periods of time between adaptation events, where the timeliness of the system can be appraised, and estimated within a single configuration of the system. Also, since the measure of a timing analysis process will be its accuracy and precision, a reasonably stable configuration of the system is required to assess the estimate against observed behaviour.

Stateless Components: Components are assumed to be stateless, in that there is no prerequisite on managing or maintaining the state of the system during an adaptation. The dynamically-

adaptable software, and to a lesser extent the adaptation process itself, are secondary to the task of estimating the volatile execution-time behaviour of dynamically-adaptable systems. The restriction on allowing only stateless components within the adaptation framework is more on simplifying the adaptation process, since the fact that the software changes at run-time (along with its execution time), is more critical than the nature of any state management during adaptations. If an adaptation framework is introduced to manage state transition between software components at run-time, there should be no impact on the timing analysis of that dynamically-adaptable system. The limitation on using only stateless components within a dynamically-adaptable system is solely a means of expediting the adaptation process, and in turn demonstrating the run-time timing analysis approach that forms the subject of this thesis.

Statistical Estimates: The overall approach towards predicting software timeliness uses a statistical-based predictive model, that is generated and updated at run-time, using timing measurements of the underlying system. The output of this predictive model in turn feeds back into a basic adaptation selection process, providing an example of one type of input into what could be a more expansive adaptation selection mechanism. Adaptations, and the adaptation management process, are limited to functional changes within a single software system. This thesis does not cover the timing analysis of any adaptations across a distributed environment, nor is there any provision for forecasting the timeliness of multiple virtual instances of a dynamically-adaptable software system operating on a single server.

Clock Resolution: The standard system clocks used in this thesis to measure the software execution times are assumed to have a clock resolution on the order of 1 millisecond, meaning that the error associated with any measurement is assumed to be no more than $\pm$ 1 millisecond. The predictive model in turn, takes these measurements as the basis for its timing estimation process, assuming that the inherent error in the measurements is negligible compared to the size of the timing intervals being measured. For applications with a sub-millisecond timing requirement, the approach outlined in the thesis would still be applicable, but the hardware used to generate the timing measurements would need to be more fine grained.

Timing Measurement: Lastly, the timing analysis approach described in this thesis assumes the

execution time of the software will vary between adaptations, as well as over extended periods of operation within a single configuration of the system, i.e., due to trends in the operating environment. The predictive model used to generate the timing estimates is assumed to execute concurrently with the software, be updated regularly, and notified by the adaptation manager whenever a functional adaptation occurs. The time interval being estimated by this predictive model is assumed to be the execution of the main control loop of the software, i.e., there is an underlying assumption that the execution path of the software is cyclical, and comparisons can be made between individual cycles. Fixing the scope of the analysis to the entire system, rather than just a sub-component, provides a more relevant assessment of overall performance, and more readily usable input into the adaptation process.

1.5 Issues Not Covered

The research question at the center of this work is how to accurately estimate the timeliness of software liable to undergo functional changes at unpredictable intervals. The dynamically-adaptable software serves only as a framework to demonstrate the predictive model outlined in this thesis. Specifically, any adaptation selection mechanism, or any other method of evaluating the optimal configuration of a dynamically-adaptable system at any given time is outside the scope of this work. Likewise, the scheduling of adaptations or any feature checking associated with the adaptation process is not considered within this thesis. Adaptations are assumed to occur rarely, and no provision is made in this work to restrict adaptation cycles from developing, nor perform any functional analysis on the adaptations themselves.

Similarly, there is no provision in this work for managing any translation of state information between software components being swapped into the system at adaptation-time. The implementation of any complex adaptation selection process is outside the scope of this thesis, excepting the assumptions described in Section 1.4. Finally, the determination of the optimal configuration of the system is not considered in itself, but only as a means of demonstrating timing feedback into the adaptation process.

1.6 Evaluation

The run-time statistical forecasting approach described in this thesis is required to provide a precise, accurate estimate of the timeliness of dynamically-adaptable software, without halting the execution of the system, relying on any detailed off-line analysis, or restricting the scope of potential adaptations. The accuracy and precision of the timing estimates produced must be sufficiently reliable and practical to enable an ongoing appraisal of the performance of the system when compared to its timing requirements. In addition to providing a qualitative assessment of the performance of the dynamically-adaptable software, the outputs of the timing analysis should inform the adaptation process, allowing pre-emptive adaptations when the software execution time is predicted to exceed a set threshold.

An evaluation of the approach outlined in this work must show how accurate execution-time estimates can be generated, at run-time, in parallel with the normal operation of a dynamically-adaptable system. The performance of the hybrid statistical forecasting approach is evaluated using a number of simulated timing behaviours, and then experimentally validated on a resource-constrained embedded device executing a time-optimizing dynamically-adaptable system. These latter timing estimates are in turn compared against the outputs of an industry-standard timing analysis tool, evaluating individual configurations of the software under ideal off-line analysis conditions.

1.7 Road Map

The remainder of this thesis is structured as follows,

Chapter 2 presents the state of the art in software timing analysis, introduces several time-predictable software design methods, and describes a number of current approaches towards enabling dynamically-adaptable software.

Chapter 3 presents the statistical approach used to forecast the execution-time performance of dynamically-adaptable software. The changeable execution-time performance of this software is described, and the methods used to generate a run-time predictive model are presented.

Chapter 4 presents the implementation of the predictive model.

Chapter 5 evaluates the performance of the statistical timing analysis approach. This evaluation

includes both a number of simulated execution-time traces, and experimental validation using dynamically-adaptable software executing on a resource-constrained device.

Chapter 6 concludes the thesis and discusses potential future work.

1.8 Summary

This chapter has introduced the fundamental concepts behind dynamically-adaptable software systems, the motivation behind their usage, and the volatile nature of their execution time. The way in which the execution time of the software both influences the adaptation process, as well as validates the performance of newly adapted software has been presented.

Since the execution time of the software forms an integral part of both performance assessment and adaptation selection, this chapter described how a predictive process is needed to form accurate, precise timing estimates without halting the system or restricting the scope of adaptations. Lastly, this chapter finishes by describing the contribution of the work, the challenges to be overcome and the various assumptions underlying this task.

Chapter 2

Related Work

In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever.

Niklaus Wirth

On the 5th February 1971, astronauts Alan Shepard and Ed Mitchell were preparing to begin their descent towards the surface of the Moon, when an unanticipated software problem prompted what was possibly one of the first instances of run-time adaptation. Controlling their spacecraft through its descent and landing was the on-board Apollo Guidance Computer, a weighty device containing just 2Kb RAM with a 2MHz processing cycle. This early computer system included various navigational software functions that were manually triggered by the astronauts at appropriate moments during the flight. However, the software responsible for guiding them to a safe landing was exhibiting some worryingly anomalous behaviour and had been noticed both by the astronauts themselves and the ground controllers monitoring the spacecraft telemetry.

Unknown to the astronauts, a tiny piece of solder, used to attach internal wiring to a switch on the spacecraft console, had become loose and was periodically setting and resetting an abort indicator bit within a memory register. If this spurious abort signal was to occur at a critical point during the descent, the software would automatically override the pilot, and move the craft into a safe orbit, effectively ending any possibility of a landing. A software fix was required that would both command the system to ignore any spurious abort signals as well as integrate with the existing code without causing any further problems. The analysis of the problem,

and the planning of a suitable adaptation to the software, had to be carried out remotely by ground controllers, and then forwarded to the astronauts to be manually keyed into the system to implement the desired change. To further complicate matters, the window of opportunity for the landing meant the software fix needed to be in place within two hours of the first detection of the error.

The process that was followed, and the eventual bit-level software adaptation that was implemented, is described in greater detail by Mindell [Mindell, 2008], and provides a microcosm of the software adaptation process introduced in Chapter 1. Timely analysis, albeit by a large number of experts already familiar with the system, coupled with remote testing on duplicate hardware and a robust design of the software, enabled a safe resolution of the error and a successful landing. Unfortunately, with the increasing complexity and responsibility of adaptable systems, the ability of a human-in-the-loop to both analyze and plan software adaptations has become more difficult, especially when the deadline on performing an adaptation is often now measured in milliseconds rather than hours [Smits et al., 2009]. Due to this time-pressure, adaptable software frameworks must often implement functional changes without waiting for the approval of software developer, or without any apriori testing on duplicate systems. Where there are a very large number of potential adaptations to system, an exhaustive static analysis of each software configuration may prove impractical. Conversely, run-time analysis is hampered by the software changing unexpectedly and with limited prior warning, as well as having only a short period in which to conduct any testing.

The principal contribution of this thesis is a method to automatically estimate the execution time of dynamically-adaptable software, without restricting the functional scope of any adaptations or halting and restarting the system to perform the analysis. Consequently, the main body of related work described in this chapter concerns the current state of the art in software timing analysis, with an emphasis on timing analysis methods applicable to dynamically-adaptable software systems. An overview of the various software adaptation frameworks is presented first, describing the ways in which adaptations may be applied to software at various stages during its design, implementation and execution. This chapter then concludes with a summary of current adaptation techniques as well as timing analysis methods, and discusses the approaches that most closely match the goals outlined for this thesis.

2.1 Software Adaptation

Software adaptation can be defined as the modification of software to suit new operating conditions. A number of adaptation frameworks currently exist, that support diverse approaches towards the design, scheduling, implementation and modification of adaptable software systems. For the purposes of this thesis, the goal of the run-time adaptation process is to optimize the execution-time performance of the software under a range of operating conditions. For example, if the system starts to exhibit a significant delay in processing tasks during periods of high load, processor-intensive software components are automatically replaced by lightweight alternatives. Within the idealized dynamically-adaptable system, introduced in the previous chapter, the adaptation process would either briefly pause execution, or hot-swap components at run-time to effect the desired functional changes. However, different adaptation frameworks each impose a slightly different set of limitations on the adaptation process, such as restrictions on when adaptations can occur and what constraints must be applied to the design of the underlying software. In order to characterize these different approaches towards software adaptation, this section evaluates current adaptation frameworks according to six basic questions concerning their motivation, focus, scope, scheduling, autonomy and implementation. These questions, initially outlined by Salehie and Tahvildari [Salehie and Tahvildari, 2009], are expanded below, and summarized as follows;

- *Why are software adaptations required?*

  The primary motivation behind software adaptations are unexpected changes in the operating environment. However, the impetus for other adaptation frameworks may include a number of functional, non-functional or system-specific issues, depending on the requirements of the system or the nature of the operating environment. Functional concerns include scheduled upgrades, run-time patches and event-driven functional updates [Oreizy et al., 2008]. Non-functional concerns encompass perceived deficiencies in the performance of the system not directly related to its functionality, e.g., the processor load, memory footprint or execution time, that necessitate corrective action in the form of a software adaptation [Sharma et al., 2004]. Lastly, system-specific issues prioritize a particular facet of the software for adaptation in order to preserve a dominant property of the system, e.g., adapting the software to resolve security concerns [Perkins et al., 2009], ensuring system stability after adaptations [Heo and Abdelzaher, 2009] or dealing with

reliability issues within highly-available systems [Zhang et al., 2005]. In contrast to the type of unconstrained dynamic software adaptation that forms the primary focus of this thesis, system-specific adaptation frameworks restrict adaptations to achieve more limited aims, e.g., improving system security against network-based attacks [Chess et al., 2003]. Unconstrained dynamically-adaptable systems may change the objective function driving the adaptation process during execution, e.g., initially adapting the software to improve timeliness, followed by subsequent adaptations to improve its memory usage.

- *Where does the adaptation take effect?*

  Software adaptations can target different logical layers within the system, depending on the adaptation framework being used. Adaptations may be applied to software at a per process, per component, per class, per function or per function-call level. Typically however, adaptation frameworks limit the scope of an adaptation action to a single layer within the system, enacting functional modifications on that layer alone. For example, component-based adaptation frameworks assume that any functional changes are restricted to the level of components [Ayed and Berbers, 2007], i.e., adaptation actions are implemented through the addition, replacement or removal of individual software components within the system. Other interpretations on where adaptations can take effect include dynamically creating processes within adaptive grid-based systems [Buisson et al., 2007], restricting adaptations to an adaptive middleware layer [Sharma et al., 2004], or intercepting and re-directing method calls [Assaf and Noyé, 2008]. In this chapter, the question of where adaptations occur within the system is used to categorize the various approaches and frameworks that facilitate software adaptation.

- *What is the result of an adaptation?*

  Adaptation frameworks may be grouped according to the extent of the functional change each adaptation can have on the underlying software. Functional modifications may discretely alter existing software behaviour or may, alternatively, replace entire parts of the system with new, updated or alternate functionality. Static composition is the most restrictive form of adaptation, since execution must be halted to implement what effectively amounts to a re-design of the system carried out off-line [Hissam et al., 2003]. Parameter-based adaptation permits limited modifications to the software at run-time [Ensink et al., 2003], but prevents any new functionality being added. Logical adaptations, and architec-

tural adaptations allow changes to be made within logical elements [Sadjadi et al., 2004], and between elements respectively [Dowling and Cahill, 2001]. Finally, dynamic adaptation permits the full range of functional additions, replacements and deletions on discrete functional elements within the system at run-time [Zhang et al., 2005].

- *When does the adaptation occur?*

  The scheduling of adaptation actions, i.e., the period when they are initially triggered, is an important consideration within adaptable software systems [Cheng et al., 2008]. While re-configuring software from an off-line context allows the software developer sufficient time to implement and test functional modifications [Wall et al., 2002], the ability to quickly react to changes in the operating environment is lost. Conversely, dynamically-adaptable software systems by their nature, react to changing conditions at run-time [Ayed and Berbers, 2007], but must perform an analysis without halting execution [Perkins et al., 2009]. Between the two extremes of off-line and run-time adaptation, it is possible to implement adaptations at other periods during the design, deployment and execution of the software, such as at compile-time [Vanderperren et al., 2005] or link-time [Popovici et al., 2003]. The question of when adaptations can occur is used to differentiate the various adaptation frameworks, according to their support of either off-line, compile-time, load/link-time or run-time adaptation.

- *Who manages the adaptation?*

  The responsibility for selecting, planning and implementing a software adaptation may be fulfilled by an external supervisor, or controlled automatically within the adaptation framework itself. A dedicated external supervisor, e.g., a software developer, provides a human-in-the-loop to oversee the adaptation process, typically re-configuring and testing the software from an off-line context [Biyani and Kulkarni, 2005]. However, an internal adaptation management function may be better placed to automatically respond at run-time to changes in the operating environment [Smits et al., 2009]. This internal adaptation manager may take the form of a policy-based rules engine [Keeney and Cahill, 2003], an algorithmic approach [Smits et al., 2009] or an internal performance model with adaptation feedback [Epifani et al., 2009]. The two categories of adaptation management considered in this taxonomy are supervised adaptation frameworks, i.e, those with a human-in-the-loop, and automatic software-controlled adaptations.

- *How is the adaptation implemented?*

  The final property used to classify adaptation frameworks is the particular manner in which functional changes are enacted on the underlying software. In contrast to the scope of a particular adaptation action, or the layer within the system at which it is applied, this property describes the method or technique used to modify the software. Adaptations may be implemented through re-compilation [Hissam et al., 2003], re-configuring the software (e.g., modifying parameters) [Sharma et al., 2004], re-arranging the existing functional architecture [Dowling and Cahill, 2001], replicating executing processes [Buisson et al., 2007], re-direction of the execution flow [Assaf and Noyé, 2008] or dynamically adding/replacing/removing software elements at run-time [Ayed and Berbers, 2007].

Taken together, these fundamental questions allow the various adaptation techniques currently in use today, as presented by a number of recent survey papers [McKinley et al., 2004] [Dobson et al., 2006] [Cheng et al., 2008] [Kell, 2008] [Salehie and Tahvildari, 2009], to be categorized according to their overall approach, functional flexibility and inherent limitations. Certain approaches, such as off-line composition [Wall et al., 2002] or re-compilation [Hissam et al., 2003], are arguably not software adaptation techniques in themselves, however they are included here for completeness. The adaptation frameworks that most closely match the definition of dynamic software adaptation given in Chapter 1, are distinguished by their support for the addition of new functionality, at run-time, without halting and restarting the system, or requiring external supervision. The various adaptation frameworks are described in this section according to their overall type, and are further analyzed according to their motivation, focus, scope, scheduling, autonomy and implementation. A comparison and summary of the various properties of each adaptation framework is presented at the end of this section, in Table 2.1.

2.1.1 Static Composition

Static composition loosely fulfills some of the characteristics of software adaptation, whereby the software is modified, albeit from an off-line context by an external agent, to better suit its operating environment. Typically, static composition is performed under the direction of the software developer, who modifies the system in response to revised software requirements, scheduled software updates, or initial performance testing. Static composition using verified software components was first proposed by McIlroy in the late 1960's [McIlroy, 1968]. He asserted

that rather than developing the functionality of each new software program *ex nihilo*, software should be created from a series of standardized re-usable software components, with clearly defined interfaces, and well-documented functional behaviours. These components could be then assembled by the software developer to produce high-quality dependable systems. The perceived maturity of more established engineering disciplines such as electronic engineering, showed the benefits of using standardized off-the-shelf components as the fundamental building blocks for developing more sophisticated devices. However, the added difficulty of composing complex software programs from basic functional components can overlook the subtle interplay between functional and non-functional behaviours within any software system. As a result, the composition of reliable systems from re-usable off-the-shelf software components is still the subject of ongoing work some forty years later [Koziolek, 2010] [Fredriksson et al., 2007] [Eskenazi et al., 2004].

The creation of complex, but predictable, software systems through static composition is described by Hissam et al. [Hissam et al., 2003]. This framework, called Predictable Assembly from Certified Components (PACC) [Ivers and Moreno, 2008], uses Prediction-Enabled Component Technology (PECT) [Wall et al., 2002] as the basis for constructing complex dependable software systems from certified software components. The overall approach combines design processes, software development tools and analytic models, to construct predictable systems from a set of certified software components. However, the focus on producing certified software components entails restricting software developers from using functionality that would make analysis of the software less predictable, e.g., by using recursive functions, run-time resolved loop conditions, or dynamic data structures. Their approach advocates predictability over software expressiveness, and commits the creation, analysis and deployment of component assemblies to be performed from a static context, i.e., off-line, under the supervision of the developer.

Similarly, Software Product-Line (SPL) development utilises a shared set of basic functional elements to statically compose a family of similar software applications. By promoting software re-use, the goal of SPL is to reduce development costs, and increase the dependability of the final software system. Typically, SPL development methods are a requirement of the application domain where a large number of different variants of the 'same' software are needed. For example, within the automotive industry, software product-line development methods aim to create vehicle control software for different vehicles, using as many of the same basic software elements as much

as possible [Thiel and Hein, 2002]. Some SPL tools provide a visualisation environment to support meta-modelling of software components [Nestor et al., 2008], allowing the software designer to browse through the hierarchy of potentially re-usable software elements, selecting components that best suit a particular task. The SPL domain readily lends itself to the use of component-based software [Pretschner et al., 2007], as well as offering the potential for dynamically-adaptable software systems to fulfill the goal of multiple software variants for different hardware instances. However, hard real-time requirements within many applications domains, and the necessity to compose and verify different configurations of the software statically [Engblom et al., 2001] [Bhylin et al., 2005], currently override any considerations about introducing run-time software adaptation within product-line software approaches.

Cadena is an add-on for Eclipse that provides an integrated modelling environment for the static composition of component-based software systems [Ranganath et al., 2003]. The meta-modelling support within Cadena allows various component frameworks such as the CORBA component model and Enterprise Java Beans, to be selected as the underlying component framework for a particular system. CALM, the Cadena Architecture Language with Meta-Modelling [Childs et al., 2006], builds upon Cadena, and supports a three-tiered meta-modelling approach to apply architectural constraints during the development of large-scale component-based software systems. CALM allows software developers to create their own bespoke modeling languages to build specific types of components, interfaces, connectors and applications. This facilitates an efficient translation between platform independent models and platform specific models, using the in-built CALM modelling tools, over a number of incremental steps. This model refinement, and the use of object-oriented concepts such as inheritance hierarchies facilitate SPL development, as application variants, resource considerations and functional limitations can be specified at various degrees of abstraction during the design process. Although CALM/Cadena supports the development of related component-based software systems, its support for subsequent software adaptation after the design process has completed is unclear. It is likely that although promoting component re-use through meta-modelling, the actual implementation of any component-based system remains fixed once deployed and executing.

The Palladio component model [Becker et al., 2009], is a meta-model for creating component-based systems with a focus on reliability and performance. The design and development of component-based applications separates the software developers who initially create the com-

ponents (component developers), from the software developers that compose the final system (component assemblers). Palladio has component developers specify and implement software components, which are stored in a component repository along with a description of the relationship between the provided and required services within each component. Applications are then formed by component assemblers taking the component information (i.e., component interface descriptions, usage models and service dependencies) and matching individual components to resource models for the target environment. The Palladio component model includes a toolkit that integrates with the Eclipse platform, as well as in-built performance analysis for the specified QoS properties of the application. It is unclear whether the component-based applications produced using Palladio permit any subsequent functional adaptation. However, since the operating environment is modelled in advance, and components are pre-assigned to resources within the system, adaptations may be prevented in order to maintain the QoS attributes of the application.

Analysis - Within statically-composed systems, the software is created to meet a set of functional and/or non-functional requirements. These requirements may precede the implementation of the software, or may evolve during software development [Kumar et al., 2008]. However, once the software has been statically-composed and deployed, further modifications to the system require execution to be halted, and the software to be re-composed off-line. Any performance information gleaned during run-time, or any functional bugs reported within the software, may be corrected only with a re-design of the software from an off-line context. Another potential disadvantage with static software composition, especially within component-based systems, is that compositional frameworks can neglect the emergent non-functional performance characteristics of software that is statically composed from many discrete software elements [Diaconescu and Murphy, 2005]. Due to inter-dependencies within many individual software elements, and differences between their usage models and what is required by the overall application, a set of reliable components does not necessarily create a reliable system [Hissam et al., 2003]. Similarly, the off-line static composition process may prove wholly unsuitable when insufficient (or incorrect) information is provided about the operating environment [Zave and Jackson, 1997]. However, for all the potential problems arising from static software composition, it does offer a number of unique benefits.

One of the principal advantages of off-line software composition is that it affords the software

developer/tester an extended period in which to evaluate the performance of the system without interfering with its execution. For example, static timing analysis techniques (see Section 2.2.1) require extended periods of off-line evaluation and testing to determine the likely execution-time of software [Sehlberg et al., 2006]. This can be achieved only if the software remains fixed during the analysis process, and the developer is afforded an opportunity to proceed with extended off-line assessment [Souyris et al., 2005]. The outputs from off-line performance analysis can be used to re-design the software to alter its functional, logical or architectural composition in order to meet its requirements. While this may produce a software system optimized towards an expected range of operating conditions, any unanticipated variation, or unexpected errors during execution cannot be addressed without further interruption and off-line intervention.

Statically-composed software is off-line when an 'adaptation' occurs, so the effects of any software modification can occur at a per component, per class or per function level. This may provide sufficient time to perform any software re-design and analysis, but it also requires someone to supervise both the 'adaptation' and the subsequent performance analysis. However, if the hardware is too complex to accurately model using static analysis techniques, or the scale of the composed system defeats any exhaustive analysis [Kirner and Puschner, 2008], a run-time approach towards timing analysis, as described in this thesis, may be more effective than current static analysis methods.

2.1.2 Compile-time Adaptation

Compile-time adaptation differs from static composition, in that the underlying software is never directly modified by the developer. Instead, adaptations to the system are designed separately, and automatically integrated into the code at compile-time, allowing new functionality to be introduced through the normal software build process. One of these automated build processes, the CiCUTS framework presented by Hill et al. [Hill et al., 2008], provides a continuous integration environment where source-code repositories are constantly monitored, with any changes triggering the software build cycle, and re-running the various unit test cases against the newly compiled code. CiCUTS uses profiling techniques to monitor the performance of the latest compilation of the software, and present a graphical analysis to the software developer. The developer can then use this ongoing performance analysis to evaluate potential modifications to the software architecture or constituent components in order to improve specified QoS metrics.

The KOALA framework [Asikainen et al., 2003], consists of a component model and specialized architecture description language (ADL) to support product line software development for consumer electronic devices. Some limited compile-time adaptation is supported through the use of source code switches within software components, that allows re-usable software to be easily reconfigured for different applications within different devices. Dhurjati et al. [Dhurjati et al., 2006] describe the SAFECode framework, a compiler-based approach towards resolving any latent memory management issues within C programs. SAFECode uses a series of additional compile-time checks to ensure that no memory allocation errors occur during execution. Traditionally, compilers apply simple lexical and semantic analysis to C source code to transform it into an executable, and leave run-time considerations, such as memory management issues, largely up to the system. In contrast, SAFECode extends the compiler to look for dangling pointers, unresolved array bounds or any calls to uninitiated pointers that could cause the run-time failure of the program. Similarly, Cooper et al. [Cooper et al., 2002], describe a framework that allows compilers to adapt their behaviour towards the type of application being developed and the target hardware environment. They describe how their framework enables adaptive compilers by providing tools to automatically configure the compilation process. The target code produced by the resulting compiler can be created to suit a particular system, as well as minimize an explicit external objective function, e.g., memory utilisation. Other techniques, such as Aspect-Oriented Programming (AOP) also extend the compiler to include additional functionality, but leave the compilation process otherwise untouched.

AOP provides a convenient method of implementing discrete functional changes across an entire codebase at compile-time. Advice code is inserted at compile-time at locations in the codebase designated by a set of join-points (pointcut). The advice code and base code are then woven together to create software with a new set of functional or non-functional behaviours. AspectJ supports compile-time adaptation, albeit through the weaving of aspect advice into the underlying base-code under the direction of a software developer [Hilsdale and Hugunin, 2004]. The AOKell component model uses the AspectJ compiler as a means of integrating updated control logic into software components at compile-time or load-time [Seinturier et al., 2006]. AOKell uses aspect-weaving to make the normally opaque and off-limits control layer accessible to software developers. Specialized control components within the AOKell framework bind to application-layer components, and provide them with naming, binding and life-cycle management

services. By modifying the control components, AOKell allows applications to be tailored for specific operating environment, e.g., self-testing or self-healing applications. While both AspectJ and AOKell restrict the modification of the base-code to changes applied at compile-time, the paradigm of AOP has been extended to dynamically weave new code into a running application by other aspect-based frameworks (see Section 2.1.4).

Analysis - Compile-time adaptation provides a more dynamic approach over static composition, in that there is a clearer separation between the original software, and the modified functionality. Adaptations are typically prompted at compile-time by revisions to the system requirements, or functional errors caught during initial testing. Since changes are more difficult to enact through re-compilation than re-composition, the scope of any change may be reduced, but the effects of compile-time adaptations are similarly limited to components, classes or functions. The results of a re-compilation are usually small-scale modifications to existing functional behaviours, i.e., logical rather than architectural changes. Although there can be a certain amount of automation to the build process, e.g., the CiCUTS approach described by Hill et al. [Hill et al., 2008], a software developer is still required to oversee the process. The adaptations or re-compilations can be implemented by directly altering the code, e.g., as with SAFECode [Dhurjati et al., 2006], or through aspect-oriented approaches, e.g., Aspect/J [Hilsdale and Hugunin, 2004].

Compile-time adaptation frameworks complement static timing analysis techniques, since both are performed on the completed source code, at the same time in the software development process. Static timing analysis tools, such as SWEET [Ermedahl et al., 2007], can be performed in parallel with the software build process, and re-run if any subsequent modifications are made to the software. Run-time timing analysis methods, in contrast, rely on the executing software, rather than the source code, to construct an estimate of software timeliness. As such, run-time timing analysis methods are unfeasible for compile-time adaptation frameworks.

2.1.3 Load/Link-time Adaptation

Load or link-time adaptation occurs when software modifications are compiled separately from the base-code, and only integrated immediately prior to execution. The JMangler framework [Kniesel et al., 2001], supports functional adaptations within Java programs at load-time, by replacing the standard Java class-loader with the Java HotSwap API. Both the base software

and the adaptation functionality are compiled as Java bytecode, and integrated at load-time using JMangler. The result is a merger of both the new and underlying functionality, effectively implementing a load-time functional adaptation.

Both PROSE [Nicoara and Alonso, 2005] and AspectJ [Hilsdale and Hugunin, 2004] enable load-time software adaptation. The software is modified by weaving additional aspect advice as the classes are loaded. Similarly, Popovici et al. [Popovici et al., 2003], describe the Just-In-Time (JIT) aspect compiler, that supports either compile-time or load-time aspect weaving within Java programs. In the latter case, the compiler can either weave minimal hooks into the base-code for later integration with the aspect code, or can use the JIT compiler to weave the advice code directly into the native code. The authors describe that while the hook-based approach offers a better separation between aspect advice and the base-code, as well as preserving the usual Java security model, it has a higher execution overhead. Each time a join-point is reached during execution, the hook method is called. Conversely, directly weaving advice using the JIT compiler may provide faster code, but violates the Java security model, and does not maintain a clear separation between the base-code and adaptations to the base-code.

The Component-Integrated Ace ORB (CIAO), provides a CORBA-compliant framework for building real-time component-based systems [Wang et al., 2004]. The Ace ORB extends the CORBA event service, to support real-time communications between components in a distributed component-based system. The middleware controlling this component assembly allows ORBs to be modified at start-up (load-time), using configuration files describing the required communications strategies for the application. Unlike aspect-based or compiler-based approaches, CIAO allows some further run-time adaptation at the process level, i.e., changing processes during execution through the middleware layer.

Analysis - Load and link-time adaptation is motivated by the same factors that prompt compile-time adaptation - unanticipated changes to software requirements during the design process, or updates due to errors or performance issues flagged during initial testing. The costs associated with a complete software re-design to these issues are much larger than a simple functional or non-functional update, implemented at load/link-time. Adaptations that occur at load/link-time may not have fine-grained access to implement changes within particular classes, e.g., JMangler [Kniesel et al., 2001], since the adaptation is enacted on pre-compiled black-box software. Instead,

re-direction may be used at load/link-time to alter functionality at a per class level. The results of an adaptation is typically the replacement of an existing logical element within the system with an alternate element, or the addition of new functionality at a specified point in the software. Adaptations occur at load/link-time, under the direction of the software developer. Finally, load/link-time adaptations are enacted on the system principally through either re-direction, or aspect-weaving.

Load/link-time software adaptation approaches lend themselves more towards static rather than dynamic timing analysis techniques. Dynamic timing analysis approaches are unsuitable when the adaptation process occurs outside of normal software execution. Static timing analysis techniques are more relevant when the analysis can be carried out from an off-line context, where there is sufficient time available for the software developer to perform a detailed evaluation of the source code. Even in the case where the source code for adaptations is unavailable, such as with black-box components, static timing analysis techniques such as MOQA [Schellenkens, 2010] can still provide an estimate of software timeliness.

2.1.4 Dynamic Adaptation

Software can be characterized as being dynamically adaptable when it allows new functionality to be added, or existing functionality to be replaced or deleted, without having to halt and restart the system to implement the adaptation. During the adaptation period, normal execution may either pause briefly while changes are made on the underlying software [Fritsch and Clarke, 2008], or instantaneously transition (hot-swapping) between different configurations of the system, as viewed by the user [Rasche and Polze, 2005]. Run-time adaptable systems typically include some adaptation management functionality, to initiate and control the adaptation process without deferring to external supervision [Smits et al., 2009]. For example, Hummel and Atkinson [Hummel and Atkinson, 2010] describe a test-driven approach towards automatically selecting the appropriate functionality within component-based systems. Within other frameworks however, the selection and scheduling of an adaptation action can be performed exclusively by an external supervisor, and require an explicit command to initiate a functional modification to the system, e.g., the IRIS framework [Sutton et al., 2006]. For run-time adaptable systems, the principal factor prompting an adaptation is variation within its operating environment, specifically, changes in the operating environment that expose a performance deficit within the current configuration

of the software.

The Chisel framework [Keeney and Cahill, 2003], provides a context-aware approach to dynamic software adaptation. Adaptations within Chisel are triggered by changes in the operating context of the system, which is defined through a combination of system resources, application and user inputs. A policy-based approach then drives the adaptation mechanism, monitors the context information coming into the system, and implements the appropriate functional adaptation at run-time using Iguana/J [Redmond and Cahill, 2002]. Keeney [Keeney, 2004], describes using the Chisel framework to support unanticipated run-time software adaptation. Unanticipated functional adaptation does not constrain the scope of any adaptation to a set of pre-defined functional behaviours, rather, it allows new functionality to be added to a running system well after its initial deployment. Adaptations are selected automatically as a reaction to changes in the operating environment, as well as the addition of new functional behaviours. In contrast, the IRIS framework [Sutton et al., 2006], provides a different approach towards adaptation selection, by leaving the process entirely under the control of an external manager, i.e., the software developer. Implementing Radio in Software (IRIS) moves much of the functionality found within wireless networking protocols from a static implementation within hardware to a purely software implementation within a reconfigurable software system. When the radio environment changes, the software developer can update an XML-based architectural model of the system, which in turn prompts a reconfiguration of the system to support a different radio networking protocol.

The analysis of a run-time adaptable system can prove problematic due to the unscheduled nature of adaptations, as well as the potentially large number of possible configurations of the software. The latter problem, a potential state explosion within even modest-sized adaptable systems, can make any exhaustive off-line testing impractical [Smits et al., 2009]. However, Biyani and Kulkarni [Biyani and Kulkarni, 2005] describe a method of simplifying the verification of an adaptable system with multiple potential configurations, by restricting adaptations to like-for-like replacement actions within a pre-defined component family. Similarly, Adler et al. [Adler et al., 2007], outline an approach towards constrained adaptation, where the complete specification of the adaptation behaviour are available at development time. Since these are available in advance, validation and verification can be performed off-line to calculate the probability an adaptive system assumes a specific configuration. Their framework, Methodologies and Architectures for Runtime-adaptive embedded Systems (MARS), allows them to analyse a closed adaptive

system, and construct a component fault tree (CFT) as well as an associated probability for each configuration of the system. The authors suggest this is a pre-requisite for development of adaptable software in the safety-critical and embedded systems domains.

Usually, aspect-oriented techniques apply changes to software at either compile-time [Seinturier et al., 2006] or load-time [Nicoara and Alonso, 2005], however, there are approaches that use dynamic aspect weaving to introduce functional changes to software at run-time. Both Nu [Dyer and Rajan, 2008] and Dynamic AspectJ [Assaf and Noyé, 2008] inject the aspect code into the underlying base-code without having to halt execution, or re-compile the system. The scope of an adaptation is defined at the level of the method-call, essentially re-directing the flow of execution at appropriate points within the program. Adaptations using dynamic aspect weaving typically add a small processing overhead [Assaf and Noyé, 2008], but facilitate the addition or replacement of functionality at run-time.

Since adaptation is more easily achieved where there are well-defined logical divisions within the software, many component-based frameworks have been developed to support run-time software adaptation. OpenCCM [Ayed and Berbers, 2007], Plastik [Batista et al., 2005], Fractal [Bruneton et al., 2006] and the K-Component model [Dowling and Cahill, 2001] all support run-time adaptation through the modification of the software components used to constitute an application. An example of a component-based adaptation framework is the TimeAdapt approach [Fritsch and Clarke, 2008], that targets software adaptation within component-based real-time systems. A specifically created reconfiguration language is used to describe the extent of an adaptation, as well as indicating any timing constraints imposed on the adaptation process itself. Since execution is suspended while the software reconfiguration occurs, the TimeAdapt framework analyses the extent of the proposed adaptation, and provides an estimate of this adaptation-period exceeding a stated time bound. The goal of a time-bounded adaptation process is to minimize interruption to normal execution by enforcing adaptations within preset time limits. An analogous approach is described by Rasche and Polze [Rasche and Polze, 2005], for a framework that supports run-time reconfigurations within real-time systems.

The Dynaco framework, presented by Buisson et al. [Buisson et al., 2005], describes an approach towards supporting dynamic adaptation within large-scale distributed computing grids. Dynamic adaptation within these grid environments focuses on maximising the performance of the distributed system by spawning new processes when resources become available. The

framework is based on Fractal [Bruneton et al., 2006], and can select the process that provides the best use of currently available resources, extend this process as resources become available, as well as automatically managing the release of resources to be reclaimed by the platform. A control loop, similar to that illustrated in Chapter 1, is used by the framework to observe, decide, plan and execute adaptations. Whereas the processes running on the grid platform can expand and contract according to resource availability, the support for functional adaptations to running processes is unknown, but presumed not to be supported.

Lastly, a number of middleware approaches currently support run-time software adaptation. The Runes middleware [Costa et al., 2007], supports dynamic component reconfiguration within network-enabled embedded systems for disaster scenarios, as well as for optimization within more resource-enabled platforms. Devices supporting the Runes middleware are capable of adapting their functional behaviours and communications strategies, based on inputs from their operating environment. While this approach supports dynamic reconfiguration of both components, and inter-component connections, as well as the deployment of new functionality, it is unclear whether the middleware provides any timeliness guarantees on the reconfiguration process itself. Sharma et al. [Sharma et al., 2004], present the QuO middleware that enables the adaptation of component-based systems by modifying the execution flow through middleware. Based on changing QoS requirements, the Quo framework adjusts the behaviour of the software by altering parameters within the system that re-direct inter-component communications. This approach, typical of adaptive middleware frameworks, is limited in that it does not support unanticipated adaptation, nor can new functionality be added to the system once it is deployed and executing.

Analysis - The main contribution of this thesis is an approach towards predicting the timeliness of dynamically-adaptable software. Although timing analysis is the principal concern of this work, the nature of the software being analysed must be taken into consideration, i.e., the six questions introduced at the beginning of Section 2.1. Whereas the previously described software adaptation methods (compile-time, link-time, etc.), provide an insight into potential implementations and limitations within the software adaptation, only run-time software adaptation provides sufficient functional flexibility to support dynamically-adaptable software systems. Run-time software adaptation forms the principal underlying adaptation mechanism for this thesis.

Software adaptations that occur at run-time are driven by changes in the operating environ-

ment, and can be initiated automatically [Dowling and Cahill, 2001], or prompted by an external user [Sutton et al., 2006]. Since there is limited time in which to implement the modifications on the software, adaptations take effect typically on a single layer within the system, such as the component-layer [Ayed and Berbers, 2007], or the middleware-layer [Sharma et al., 2004]. Adaptations can add, replace or delete functionality during normal software execution [Fritsch and Clarke, 2008]. Adaptations are implemented at run-time, but can either briefly suspend execution to implement the adaptation [Fritsch and Clarke, 2008], or immediately switch between configurations of the software [Rasche and Polze, 2005]. Usually the adaptation is managed automatically by functionality within the adaptable system, however there are some frameworks that support user-prompted run-time adaptation [Sutton et al., 2006]. The adaptation can be implemented in a variety of ways, such as replacing software components [Batista et al., 2005], adapting the middleware [Sharma et al., 2004], or run-time aspect weaving [Assaf and Noyé, 2008].

2.1.5 Summary of Adaptation Frameworks

From the perspective of this thesis, the software underlying the dynamic timing analysis approach must support both unanticipated run-time adaptation, as well as permit previously unenvisaged functionality to be added to the system during execution.

Statically composed software systems, such as PECT/PACC [Ivers and Moreno, 2008], are assembled from a closed set of functional elements. While this allows the software to be analysed, or constructed in such a manner as to be inherently predictable, it also limits the flexibility and adaptability of the system at run-time. Similarly with parameter-based adaptation [Sharma et al., 2004], the functional scope of the software is fixed once deployed to the target hardware, and may be reconfigured dynamically, but cannot be extended. As pointed out by McKinley et al. [McKinley et al., 2004], parameter-adaptation may allow systems to be tuned, but cannot allow an application to implement behaviours conceived following the initial construction of the system. Static, load/link-time and parameter-based adaptation frameworks all constrain the functional extent of the software to a potentially large, but statically analysable, set of configurations of the system. Although a large 'search space' may exist for these functionally-restricted systems, once any exhaustive timing analysis has been performed on the system, newer functionality cannot be subsequently added that would invalidate the analysis.

Approach	Why?			Where?					What?					When?				Who?		How?					
	Functional Adaptation	Non-Functional Adaptation	System-Specific Adaptation	Process	Component/Object	Class	Function	Function-call/Aspect	Static Composition	Parameter Modification	Logical Adaptation	Architectural Adaptation	Dynamic Adaptation	Off-line/Design-time	Compile-time	Load/Link-time	Run-time	Supervised	Automatic	Replication	Re-Compile	Re-configure	Re-arrange	Re-direction	Add/Replace/Remove
	Static Composition																								
CALM/Cadena	✓	✓			✓				✓					✓				✓			✓				
PACC/PECT	✓				✓				✓					✓				✓			✓				
Palladio		✓			✓				✓					✓				✓			✓				
	Compile-time Adaptation																								
AOKell			✓					✓			✓				✓			✓			✓				
Aspect/J	✓							✓			✓				✓			✓			✓				
CiCUTS	✓	✓				✓	✓			✓	✓	✓			✓			✓			✓				
Cooper et al.	✓					✓	✓				✓				✓			✓			✓				
KOALA	✓					✓	✓				✓				✓			✓			✓				
SAFECode			✓			✓	✓				✓				✓			✓			✓				
	Load/Link-time Adaptation																								
CIAO/TAO	✓				✓						✓	✓				✓		✓					✓	✓	
JMangler	✓					✓	✓				✓					✓		✓						✓	
JIT aspects	✓							✓			✓					✓		✓						✓	
PROSE	✓							✓			✓					✓		✓						✓	
	Dynamic Adaptation																								
Chisel	✓				✓								✓				✓		✓			✓	✓	✓	
Dynaco		✓		✓									✓				✓		✓	✓					
Dynamic Aspect/J	✓							✓					✓				✓		✓					✓	
Fractal	✓				✓								✓				✓		✓			✓	✓		
IRIS	✓					✓				✓	✓	✓					✓	✓				✓	✓		
K-Component	✓				✓						✓						✓		✓			✓			
MARS			✓		✓								✓				✓	✓							✓
Nu	✓							✓					✓				✓		✓					✓	
OpenCCM	✓				✓							✓	✓				✓		✓				✓		✓
Plastik	✓				✓						✓	✓					✓		✓			✓	✓		
QuO		✓			✓					✓							✓		✓					✓	
Runes	✓				✓								✓				✓		✓						✓
TimeAdapt	✓	✓			✓								✓				✓		✓						✓

Table 2.1: Summary of adaptation approaches.

Run-time adaptable systems may be distinguished from the other adaptation approaches, in that new functionality may be added to the system at any time during its execution [Batista et al., 2005]. Unconstrained run-time adaptation allows the greatest level of functional expressiveness, however the constantly changing functionality makes static (exhaustive) analysis unfeasible [Smits et al., 2009]. Both Dynamic aspect weaving [Dyer and Rajan, 2008], as well as component-based techniques [Ayed and Berbers, 2007] provide convenient methods of adding new functionality to an executing system, however there are some research challenges that still exist in selecting the appropriate adaptation [Hummel and Atkinson, 2010], and avoiding feature interaction problems with the software [Huang et al., 2008].

2.2 Timing Analysis

Timing analysis is the process of evaluating software in order to generate an estimate of its likely execution time, once deployed and running on a target hardware platform. Traditionally, timing analysis has focused on determining the worst-case execution time (WCET) performance for software, typically within real-time and embedded systems. The WCET bound that is derived provides a guarantee about the safe performance of the software, i.e., that a given software task will complete execution within a set deadline. The deadlines for the various software tasks are used to create a static processing schedule for the real-time system, allocating a set period to the completion of this task [Holsti et al., 2008]. If a task were to exceed its deadline, the execution schedule would be invalidated, and potentially lead to an irrecoverable system failure.

The determination of a safe non-pessimistic WCET bound provides a guarantee about the correct performance of the software once deployed and executing. The majority of timing analysis methods currently available are concerned with establishing the worst-case performance of software [Wilhelm et al., 2008], mostly within the context of real-time and embedded systems. However, WCET bounds are intended to encapsulate a rarely occurring event, and provide no insights into the average execution time performance of the software. Where simple QoS considerations take precedence over worst-case performance, such as in soft real-time systems, the average-case execution time (ACET) provides a more useful metric in describing the expected performance of the software [Schellenkens, 2010].

Predicting the execution time of software (both average and worst-case) is currently achieved through either static [Malik et al., 1997] or dynamic timing analysis techniques [Bernat et al.,

2003]. Static timing analysis techniques directly evaluate the software, or a representative model of the software, from an off-line context, and are further described in Section 2.2.1. However, static timing analysis techniques require the software to remain fixed while testing completes, and as a consequence, are unsuitable for dynamically-adaptable software systems liable to unanticipated functional modification. In contrast to more static methods, dynamic timing analysis techniques gather representative software timing measurements from an executing system, and evaluate this data to infer the likely average-case [Schellenkens, 2010] or worst-case [Hansen et al., 2009] timing behaviour. Measurement-based timing analysis methods can be performed fully at run-time, or both off-line and at run-time, depending on when the evaluation of the timing trace data is performed.

Both static and dynamic timing analysis methods can be further categorized according to their inherent limitations, implementation and overall goals. By modifying the six fundamental questions outlined earlier in Section 2.1 (e.g., why, where, what, when, who and how), the various approaches towards software timing analysis can be sorted into more fine-grained categories. These classifying questions can be outlined as follows;

- *Why is software timing analysis required?*

  Timing analysis is used to produce estimates about the likely execution time of software, running on a specified hardware platform. This need for a timing estimate may be prompted by requirements within the particular application domain, e.g., safety requirements within real-time and embedded systems, or merely questions about the likely performance of the software. The reason why timing analysis is performed in the first instance may be classified as either assuaging safety concerns for real-time systems, testing the software during the design process, or validating the performance once deployed to the target hardware environment.

- *Where is the timing analysis applied?*

  Timing analysis can be applied directly to the software during the design process [Eskenazi et al., 2004], to abstract models of the software or hardware [Li et al., 2007], or timing trace information recorded during execution [Wenzel et al., 2005]. Various timing analysis techniques can be applied depending on the information currently available about the system, the state of the software (completed/in development), as well as the time available in which to carry out the analysis. For this taxonomy, the timing analysis is said to be

applied either directly to the software, to abstract models of the software, or timing trace data of the executing software.

- *What is the result of the timing analysis?*

  The output of a timing analysis process may focus on either the limits of the likely execution time, e.g., the best-case/worst-case behaviour [Wilhelm et al., 2008], or may alternatively provide an estimate of its average-case behaviour [Schellenkens, 2010]. Typically, the various analytic tools and methods applied to determine software execution times usually mean one or the other of the extreme or average-case timing behaviours is the principal focus of the approach. Both approaches are not mutually exclusive, but simply selected as required for the particular operating environment. For example, the approach described in this thesis, and outlined in the next chapter, provides both average-case and worst-case timing estimates. Within this section, timing analysis approaches are classified according to whether they are setup to provide extreme-case (best/worst) or average-case timing estimates.

- *When does the analysis occur?*

  The majority of timing analysis methods are currently applied off-line prior to deployment, during normal software acceptance testing [Wilhelm et al., 2008]. This is due to the requirement for the software to be in a final state, so that any timing estimates produced from the source code are representative of the performance of the deployed system. However, if the software is adaptable, it does not have a single final state that can be tested before deployment. In this case, automatic timing analysis is required, where updates to the timing estimate for the system are made at run-time, under the supervision of functionality within the system [Epifani et al., 2009]. The classification of timing analysis approaches presented here considers two broad types of timing analysis technique - off-line and run-time.

- *Who supervises the analysis?*

  The timing analysis method applied to a particular system can be considered to be either supervised, or automatic, depending on whether someone is required to manage the analysis process [Ermedahl et al., 2007], or whether the system itself can perform a self-analysis during run-time [Epifani et al., 2009].

- *How is the timing analysis achieved?*

There are a number of different techniques used to estimate software timeliness, such as formal modelling [Beltrame et al., 2001], timing trace analysis [Burguiére and Rochange, 2006], flow analysis [Li et al., 2007], code analysis/simulation [Sehlberg et al., 2006] and statistical methods [Hansen et al., 2009]. These broad categories are not mutually exclusive within individual approaches, e.g., Wenzel et al. [Wenzel et al., 2005] outline a hybrid timing analysis approach that utilises both code analysis and measurement-based analysis to derive a worst-case execution time bound.

2.2.1 Static Timing Analysis

Static timing analysis techniques predict the likely execution-time of software from an off-line context, using "information collected at or before compile-time" [Malik et al., 1997]. This type of timing analysis may reason over abstract models of the system, or alternatively may directly evaluate the source code. Static analysis techniques, as they are performed from an off-line context, have a common requirement on being supervised by a domain expert, e.g., software developer or tester. Typically, the timing analysis is performed rigorously on a fixed code base over a prolonged period, until there is a high confidence that the estimate produced corresponds to the actual behaviour of the software once it is deployed and executing. Souyris et al. [Souyris et al., 2005], provide the example of a software task within an avionics system that required upwards of 12 hours of analysis to produce a (worst-case) timing estimate.

2.2.1.1 Static Analysis Frameworks

The MESCAL framework, introduced by Chen et al. [Chen et al., 2001], provides a combined software development environment and static analysis framework for the creation of embedded systems software. The analysis is performed using an Integer Linear Programming (ILP) approach, with the control flow analysis, branch prediction and predication analysis performed automatically by the framework at compile-time. The analysis can target a number of different hardware environments, by modeling the architecture separately, and sharing this information using an XML-based MESCAL architecture description. However, as typical with static analysis approaches, the MESCAL framework constrains the software in order to perform the analysis, e.g., no dynamic functions are supported, nor are the effects of interrupts or preemption considered on the performance of the software.

The CiCUTS framework [Hill et al., 2008], introduced previously in Section 2.1.2, performs some run-time QoS validation on the timing behaviour software developed within a continuous integration environment. When a change is detected in the software repository, the build cycle is executed, followed by a set of unit tests to validate the timeliness (if required) of the newly compiled software. The timing analysis is performed automatically, but the precise manner in which the analysis is performed is unclear. However, it is assumed any appropriate automatic static timing analysis tool could be used.

The MOQA framework [Schellenkens, 2010], provides a domain-specific programming language called MOQA-Java [Townley et al., 2009] that can automatically evaluate the average-case execution time (ACET) of software. The ACET bounds are determined by tracking and combining the distributions of the basic components within a MOQA program. Distributions are simple statistical models representing the expected timing behaviour of a component, however it is unclear from the paper whether the authors use the standard Normal (Gaussian) distribution, or can fit more exotic distributions to the components, e.g., the Poisson, Weibull, Cauchy or Log-Normal models. Since programs are ordered compositions of these basic components, MOQA attempts to track and control these distributions, adjusting them based on their usage, and on the notion of random bag preservation. The average-case time for the entire program can then be computed from the times of its constituent parts.

Analysis - Static analysis frameworks, while providing software developers with a convenient process to repeatedly test software timing behaviour during development, are insufficient to analyze the execution time of dynamically-adaptable software. The number of potential configurations of the system is typically too great to perform any detailed exhaustive analysis, to expose the likely execution time effects of every possible adaptation [Smits et al., 2009].

Static analysis frameworks mostly target the timing performance of the software from a view of ensuring an acceptable QoS, rather than providing any real-time guarantees. Since the frameworks are used during design-time, source code provides the basis for any timing analysis. Average-case rather than worst-case bounds are produced off-line, typically through some automated timing analysis process. The code itself is usually analysed to produce the final estimate of software timeliness.

2.2.1.2 Formal Methods Analysis

Formal methods analysis decomposes software into abstract statements or properties that can be reformulated and reasoned about algebraically. The use of formal analysis within software engineering is not new, and has been previously applied to compiler-design, software requirements analysis and describing the interactions within concurrent systems [Hoare, 1978]. Formal methods analysis applied to software timeliness is not an empirical analysis approach, rather a means of verifying established existing timing properties, under a set of basic assumptions. Without detailed knowledge concerning the timing behaviour of the software, or any safe dependable means of combining functions in a time-assured manner, formal methods techniques will encounter difficulties evaluating average-case or worst-case timing bounds.

Zhang et al. describe a modular model-checking approach to verify the behaviour of dynamically adaptable systems [Zhang et al., 2009]. They separate the functional part of the software from the adaptive logic, and model the dynamically adaptive software as a collection of non-adaptive programs with transitions between each program representing an adaptation. Using three levels of system abstraction, i.e., high-level requirements, models of adaptable components, and descriptions of the low-level implementation, the authors apply a specialized process-algebra to ensure that particular global invariants are maintained, and desired functional and non-functional properties of the system remain unaffected by adaptations. However, it is unclear how the authors establish the desired execution time bounds initially, nor is it described how unrealistic timing requirements are handled using this approach.

AMEOBA-RT is a run-time monitoring and verification technique for dynamically-adaptable software systems [Goldsby et al., 2008]. The software is modeled as a series of steady-state programs, roughly corresponding to individual configurations of the system, with adaptation actions modeled as transitions between these steady-state programs. These models are expressed using Linear Temporal Logic (LTL), as well as an adaptation-specific extension to LTL called Adapt-operator extended Linear Temporal Logic (A-LTL). AMEOBA-RT in effect acts as a run-time model-checker for software expressed using A-LTL and LTL semantics. Since adaptations are triggered by changes in the run-time state of the system, this state is monitored by instrumenting the software using AspectJ. While the analysis of the A-LTL/LTL semantics is carried out automatically, the developer must manually specify methods (pointcuts) to instrument and monitor the code using aspects. AMEOBA-RT examines only one execution-path at a time, ef-

fectively avoiding the problem of state explosion within the model-checker, i.e., an exponentially large number of potential software configurations requiring analysis. The authors illustrate the performance of AMEOBA-RT by analysing an adaptive Java pipeline program, that changes between synchronous and asynchronous behaviour depending on the prevailing CPU load. The A-LTL/LTL semantics describing the pipeline is forwarded to a dedicated model-checking server for analysis, and either the current configuration of the program is validated, or an exception is recorded in an error log for that is processed off-line.

Marref and Bernat [Marref and Bernat, 2008] describe the use of Constraint-Logic Programming (CLP) to express the constraints governing the execution flow and times of basic functional blocks within a program. Each block has an associated execution count describing the number of times it is called during execution, as well as an execution time. A WCET-bound can be found by solving an Implicit Path Enumeration Technique (IPET) model of the program, in effect, finding the aggregate of the worst-case times for each block. Both the execution count, and execution time constraints for the blocks can be found using timing trace analysis of the program, whereas the times can be further refined by performing a dependency analysis between the blocks. IPET does not consider the flow of execution between these basic blocks but instead a set of blocks with their respective execution counts. However, the CLP approach does not consider the feasibility of a particular execution path, meaning that the WCET bounds produced using this method may be overly pessimistic.

Analysis - The level of abstraction required to model a dynamically-adaptable system using formal methods makes any accurate assessment of its likely execution time unlikely, unless there is a detailed formal model of the underlying hardware, or a run-time dynamic analysis process. Several tool-based approaches have detailed processor models, however these are not specified according to any CSP-like [Hoare, 1978] or LTL-like [Goldsby et al., 2008] modelling language. Formal methods analysis provides a level of abstraction that may be useful in describing the composition or constraints within a dynamically adaptable system. However, the intricacies of evaluating the execution time of adaptable software executing on a complex processor architecture, within a highly variable operating environment, may require formal methods techniques to model the software behaviour at a very fine-grain, potentially introducing further complexity into the analysis process.

Formal methods analysis is traditionally used to provide a guarantee about the timeliness of software, however, its difficulty in forming WCET bounds without a detailed understanding of the hardware means the timing guarantees produced are concerned only with overall performance. Abstract models of the software, rather than source code or timing trace data are generally used, and applied to the estimation of either ACET or WCET bounds. Formal methods techniques are performed off-line under the supervision of a domain expert, and typically operate using process algebras to represent the desired properties of the system.

2.2.1.3 System Models

System models represent a class of approaches where abstract models associated with the performance of the software are created, and occasionally updated at run-time, to provide an indication of software timeliness. Unlike tool-based analysis methods, there is usually no model of the underlying hardware, simply an inter-connected series of parameters and properties describing the performance of the system. Also, in contrast to formal methods analysis, system modelling techniques are generally at a lower level of abstraction, and provide a different reasoning process, relying less on process algebras or model checkers.

Epifani et al. [Epifani et al., 2009], describe the Keep Alive Models with Implementations (KAMI) framework, that provides an updated model of the non-functional properties of run-time configurable software systems. Relying on apriori estimates of various parameters used to predict performance, e.g., request rate, processor load, mean response time, can be prone to errors. For example, the initial parameters used to generate a predictive model of the performance of the system may be different from the values actually experienced at run-time. In addition, the initially accurate performance model can become less representative of the system as the inputs into the predictive model diverge from the actual values over time. To provide a self-updating predictive model, the authors introduce a Bayesian analysis of the parameters and probabilities used to form the models. Using Discrete Time Markov Chains(DTMC) as well as Queueing Networks(QN), a model of the performance of the system can be produced, and evaluated at run-time. Continually updating the parameters allows the models to evolve as the software executes. Bayesian estimation theory is used to refine the parameters that provide inputs in the DTMC and QN models of reliability and performance respectively. Since software engineers are required to create and deploy models of the system to the KAMI framework, it

is unclear how a dynamically-adaptable software system could be effectively modelled, i.e., the state transitions specified in the DTMC and QN models could be quite large, depending on the number of potential configurations of the system. The DTMC and QN models are evaluated by third party model checkers incorporated into the KAMI framework, the PRISM model checker and the JMT workload analyser respectively. The authors describe the operation of the KAMI framework using the running example of a web-service composition, i.e., a decentralized web-service co-ordinated using BEPL. The goal of the web-service composition is to meet its global QoS requirements, by adjusting the models corresponding to the current configuration of the service, allowing system modellers to perform software reconfigurations as violations or exceptions are raised by the framework. The overhead of running the KAMI framework is not described in the paper, however the PRISM and JMT model checkers are not lightweight components specifically designed for embedded or resource-constrained devices. In addition to the overhead on the system, the expected time required to process the performance models is not set out within the paper. It is unclear whether an unrestricted dynamically-adaptable system, capable of altering its performance unexpectedly at run-time, lends itself to the detailed modelling approach set out in this work.

Calinescu and Kwiatkowska [Calinescu and Kwiatkowska, 2009] present an approach towards managing adaptations within autonomic systems through the run-time analysis of probabilistic models of the systems performance. Specifically, the authors describe the use of Markov chains as a means of modelling software behaviour, and use the PRISM model checker [Hinton et al., 2006] to evaluate these models at run-time. The autonomic/adaptation manager is used both to analyze the current state of the system, as well as plan parameter-based adaptations to the functional elements within the system. It appears that adaptations are limited to configuring the existing software, rather than dynamically adding new software elements at run-time. Various policies (action/goal/utility) are described, and used as the basis for both quantitative analysis and software optimization. Two case studies are presented, a dynamic power management system for a simulated disk drive, and an adaptive load balancer that dynamically assigns servers to a cluster within a data center. In the former case, the evaluations were performed at 10-second intervals, taking “a sub-second” time to complete on a dual-core 3GHz system. The later case, being a more complex model, took the PRISM engine “up to 30 seconds” to complete the evaluation of the clustered servers. These lengthy processing times would appear to make the approach

unsuitable for the complex changeable domain of dynamically adaptable systems. Also, it is unclear whether the same evaluation process could be realistically applied within more resource-constrained hardware.

Hissam et al. [Hissam et al., 2008], outline a measurement-based timing analysis approach, for a highly-configurable real-time system. The authors describe a predictive model used to estimate both the worst-case and average-case execution time of threads running on top of a real-time VxWorks operating system. The goal of their approach was to generate timing deadlines based upon settings in the configuration table for the software, timing measurements of the software components that comprise the system, and a representation of the dependencies between these components. Their approach is based on timing measurements, but still derives abstract models of the operating system, the application software and any other (middleware) software dependencies to generate the final timing estimate. Whereas their average-case predictions were within 0.8% and 1.3% to the two real-time tasks they evaluated, their worst-case prediction was more than double the observed worst-case performance for both tasks. It is unclear whether the pessimism inherent their worst-case model was deliberate, or whether their acceptable probability for exceedance was set very high, e.g., 10^{-12}.

Analysis - System models abstract the system into a series of parameters or configuration options, that can derive the performance of the software, or alternately allow various settings to be altered to enforce a particular set of timing requirements. The KAMI framework described by Epifani et al. [Epifani et al., 2009] provides a good example of this latter approach. The principal advantage of system modelling approaches towards timing analysis, is that the range of each possible parameter within the system is known in advance. This allows an exhaustive analysis to potentially expose the timing behaviour of every likely configuration of the system, or if this is too involved, a more expected range of operating conditions can be evaluated instead. However, system modelling techniques do not allow any functionality to be added to system at run-time, except with difficulty. The length of the evaluation periods required for many system modelling approaches [Calinescu and Kwiatkowska, 2009], preclude their use within systems with immediate timing requirements, such as dynamically-adaptable systems.

System modelling techniques are used primarily to maintain a specified QoS with respect to timing, however Hissam et al. [Hissam et al., 2008], consider worst-case timing bounds for real-

time systems. Abstract models of the system, usually composed of a collection of configuration options, provide the basis for the timing analysis, which can be carried out either off-line or at run-time. The analysis itself is typically an evaluation of the current state of the system, using Markov chains or an evaluation of queueing models representing the call flow graph of the system.

2.2.1.4 Tool-Based Timing Analysis

Tool-Based timing analysis applies dedicated timing analysis toolkits to the evaluation of software timeliness. Typically, timing analysis tools examine the source code to extract execution flow information, loop bounds, and code segments that are analysed against models of the target hardware environment.

The aiT tool [Sehlberg et al., 2006], is a commercial WCET analysis tool, widely used within real-time software engineering projects [Wilhelm et al., 2008]. aiT analyzes the binary executable, extracting the call flow from the object code, and performs a low-level analysis of the program executing on a detailed model of a processor architecture. Typically, the architectures supported by aiT, as well as other simulation-based analysis tools, are restricted simplistic processors with well-known behaviours, with none of the hardware optimizations usually found in more advanced CPUs. aiT analyses both the cache and pipeline effects on the basic blocks identified by the call flow analysis, and combines the analysis of these basic segments in to a WCET bound for the program.

Chronos [Li et al., 2007], is an open-source worst-case analysis tool for C programs. As with aiT, the WCET bound produced by Chronos is determined through a combination of source code analysis, and fine-grained modelling of the underlying processor architecture. A dedicated tester is required to provide input during the analysis, since although Chronos performs an initial flow analysis on the software, any unresolved loop bounds must be provided manually. However, unlike other tool-based analysers such as Heptane [Colin and Puaut, 2001], Chronos supports out-of-order pipelines and global branch prediction within the target processor. The C program is broken into individual program segments that are analysed separately. This analysis is performed using an in-built ILP solver. In essence, the WCET for each program segment is generated by finding the execution count of each segment, and multiplying this by the worst-case performance of the segment. The summation of these individual WCET values for all the basic

blocks combined then forms the final WCET value for the program.

Bernat et al. [Bernat et al., 2003], present the pWCET toolkit for the evaluation of worst-case bounds within the domain of real-time systems. pWCET combines static and measurement-based analysis techniques to provide a probabilistic worst-case bound. The program is broken up into basic blocks, using a syntax tree representation of the program. Timing trace data is then used to determine the probability distributions for the execution time of each block. These timing traces allow the pWCET tool to be platform independent, i.e., they can be derived either from cycle-accurate CPU simulators, or observations of the software running on the real system. The probabilities are then combined to provide a WCET value within an upper and a lower bound. The RapiTime commercial timing analysis tool is a direct result of the work done on the pWCET tool [Wilhelm et al., 2008]. RapiTime improves on the path analysis within pWCET, as well as allowing the user to add more extensive annotations within the code to capture loop and branch information during the measurement phase of the analysis [Mezzetti et al., 2008].

The SWEdish Execution Time tool (SWEET) [Ermedahl et al., 2005], is a modular WCET tool that performs source code analysis using models of a number of low-level processor architectures, e.g., the ARM9 processor. Since SWEET requires a model of the underlying hardware, as well as access to the source code, its application is typically limited to providing worst-case bounds for real-time and embedded systems. Similar to other static timing analysis tools, such as aiT [Sehlberg et al., 2006], SWEET calculates the worst-case bound by performing an initial flow analysis on the program, then an evaluation of the identified basic code segments against a detailed model of the process, followed by a calculation of the worst-case from this simulation. Unlike RapiTime or pWCET, SWEET can cope with recursive functions within C programs, however since it uses an Implicit Path Enumeration Technique (IPET) within its initial flow analysis, the program must be well structured to facilitate analysis.

Analysis - Tool-based timing analysis is the most widely applied method in determining WCET bounds for real-time and embedded software [Wilhelm et al., 2008]. Detailed models of the underlying processor architecture are typically used to simulate the performance of code segments, however this approach, while providing a detailed, cycle accurate estimate, is limited to very basic systems. Complex processors are difficult to model in great detail, and as a consequence, are generally unavailable within WCET analysis tools. Similarly the software being analysed may be

required to have its source code available, and restrict the use of recursive functions, or dynamically assigned loop and branch conditions. Usually the tester/software developer is expected to provide annotations, when static analysis fails to determine the loop or branch conditionals. However, aside from the workload placed on the tester, and the lengthy period required for testing, tool-based analysis remains the most dependable means of evaluating worst-case bounds within safety-critical systems such as avionics [Souyris et al., 2005] or automotive [Sehlberg et al., 2006] systems.

Tool-based timing analysis techniques are concerned with the evaluation of WCET bounds to providing timing guarantees for real-time and embedded systems. Mostly source-code analysis is used, however several tools [Bernat et al., 2003] [Mezzetti et al., 2008] include timing measurements within their analytic process. The lengthy periods required to perform the analysis, typically in the order of several hours [Staschulat et al., 2006], and the requirement to occasionally annotate the code require tool-based analysis to be performed off-line and supervised by a tester/software developer.

2.2.2 Dynamic Timing Analysis

Static timing analysis techniques must typically model the underlying hardware in order to produce a valid timing estimate of the executing software. However, as CPU architectures have become increasingly complex, the hardware optimizations built into the processor, e.g., cache behaviour, instruction pipelines and branch prediction, have rendered the task of static timing analysis more difficult. Edwards and Lee [Edwards and Lee, 2007] state that due to this complexity, the timeliness of software executing on modern processors is 'virtually unknowable'. Also, since run-time functional adaptations can unintentionally alter the execution time of software, and invalidate any timing estimates formed from an off-line context, applying static timing analysis methods to dynamically adaptable software is only suitable where every potential configuration of the software can be analysed in advance. However, static approaches are impractical since even modest-sized adaptable systems can have a very large set of potential configurations [Smits et al., 2009]. Within a dynamically-adaptable system, the order of scheduling adaptations cannot be anticipated, since they are reactive events prompted by changes in the operating environment. Where there is the potential for unconstrained adaptation, either in its scheduling or functional scope, the application of static timing analysis techniques is unworkable. Rather than basing

any analysis on simulating the software on detailed models of the underlying hardware, dynamic analysis techniques may use the run-time observations of the software to predict its likely future behaviour. Where the software itself introduces added complexity through run-time adaptation, dynamic measurement-based approaches offer a favourable means of evaluating the software's volatile timeliness.

Due to the limitations of exhaustive static timing analysis applied to dynamically adaptable software, a more appropriate timing analysis method would focus on evaluating only the current configuration of the software, i.e., the configuration resulting from a proposed adaptation during run-time. Since the system continues execution immediately after an adaptation completes, this timing analysis must be performed at adaptation-time or immediately when the software resumes execution. However, current dynamic timing analysis methods rely on a large number of timing measurements, coupled with an extended period of subsequent off-line analysis, to predict the likely execution time of the software [Hansen et al., 2009]. While these measurement-based approaches rely on empirical analysis techniques, and can cope with changeable timing behaviour in the underlying software, they still require an extended period to both collect and evaluate the timing measurements, from a run-time and off-line context respectively [Petters et al., 2007].

2.2.2.1 Measurement-driven Analysis

Measurement-driven analysis approaches are a hybrid of static and measurement-based techniques. Typically, a static analysis decomposes the program source code into basic program segments using a call-flow analysis. These program segments are small sets of instructions, usually the code encapsulated within a loop structure, branch or function. A purely static analysis technique would assign timing properties to these program segments and submit the resulting model to an ILP solver to derive a timing bound. However, the timing values may not be representative of the actual behaviour of the software executing on the target hardware environment, either due to incomplete processor models, or overlooked dependencies and execution flows within the software itself. Measurement-driven timing analysis can counteract any deficiencies in static analysis, by generating test-data (timing test cases) that execute on the target hardware, and iterates through each execution path, thereby exposing the timing behaviour of the previously identified program segments. Since measurements are used to provide timing information, hybrid static/measurement-based techniques are easily retargeted to most hardware environments, since

no model of the processor is maintained within the analysis method.

Colmenares et al. [Colmenares et al., 2008], present a measurement-driven timing analysis approach called APS Analyzer, that utilizes both run-time measurements and static analysis techniques to derive WCET bounds on C++ programs. The authors focus on applying this method to determine the execution safety of real-time distributed computing systems, by generating a safe, non-pessimistic WCET bound. Their approach breaks the software into acyclic-path segments (APSs), discrete functional blocks within the program containing no loops, and measures the worst-case performance of each APS individually. A graph representing the control flow between APS's is derived off-line, and the timing measurements for each APS is assigned to their respective node in the graph. An ILP solver is then applied to the graph and a suitable WCET bound generated for the software. This timing bound will fall between the maximum observed execution time, and an overly pessimistic static estimate. However, since the execution time measurements are generated using large randomly-generated data sets, it is unknown whether the timing trace data generated contains the actual worst-case behaviour, i.e., there may be unobserved worst-case behaviour using this approach.

The Model-based Development of Distributed Embedded Control Systems (MoDECS) framework [Wenzel et al., 2005], is a hybrid timing analysis framework, that combines dynamic and static analysis techniques to establish a worst-case bound for C programs. The motivation for MoDECS is timing analysis within safety-critical systems, and the authors describe how a significant proportion of automotive breakdowns are ultimately caused by problems in embedded systems stemming from timing issues. In systems with fine-grained or precisely-ordered task deadlines, any timing delay at best hampers the ability of the software to function as expected, and at worst leads to the complete failure of the system. MoDECS performs an initial static analysis on the software, breaking it up into basic code segments. Test data is automatically generated for the overall program using an 'evolutionary algorithm' that highlights all the executions through the program, and exercises all the program segments found using the static analysis. Each path is executed and the timing measurements for each code segment is recorded. The final WCET bound is produced by combining the execution time measurements for each segment and determining the worst-case path.

Analysis - Measurement-driven analysis attempts to solve some of the problems within static

timing analysis frameworks and toolkits, i.e., the difficulty in exposing the actual run-time behaviour of the software, and the limitations when analysis is restricted to a small number of detailed processor architectures. Hybrid, or measurement-driven, analysis techniques allow detailed execution time bounds to be derived regardless of the complexity of underlying processor architecture. However, measurement-driven analysis typically requires an initially derived call-flow graph, and a set of test data that can exercise each potential path within this graph. Where the number of execution paths is very large, a lengthy period will be required to observe and record the execution time of each path. While exhaustively testing the various execution paths may take some time, it does not necessarily expose any state information within program segments that could result in a worst-case execution time [Wilhelm et al., 2008]. Consequently, measurement-driven analysis techniques may not have the same safety guarantees as static analysis tools or frameworks.

Measurement-driven analysis methods aim to produce WCET bounds for real-time systems, operating within either embedded or more familiar desktop hardware environments. Both source-code analysis as well as timing trace data contribute to the timing analysis, which is performed off-line under the supervision of a dedicated tester (although the timing measurements are recorded at run-time). A combination of flow analysis and timing trace analysis is used to produce the final timing estimate, however some approaches can include additional code analysis [Deverge and Puaut, 2005].

2.2.2.2 Statistical Analysis

Statistically modelling complex software timeliness shifts the focus away from the cause-and-effect investigations used to derive timing predictions with traditional methods, to a more measurement-based process applied to the software in situ. The complexity of current processors, and the subtle interplay of execution-time effects, e.g., caching, pipelining, pre-fetching etc. all produce timing effects that are difficult to determine statically, even when the software itself remains unchanged after deployment. As the complexity of the underlying systems increases, establishing the timeliness of the software becomes more difficult, and it has been cited as beginning to overwhelm the capability of many formal methods-based approaches [Schmidt, 2007].

A statistical-based approach to timing analysis has been described by Edgar [Edgar, 2002].

Using statistical modelling techniques, he infers the WCET bounds of software tasks, and provides these timing bounds as inputs into an off-line scheduling analysis for real-time software. The measurement-based approach he describes relies on creating an initial call-flow graph identifying the worst-case execution path for the software, and then repeatedly measuring the execution time of this path. Once a sufficient number of measurements have been generated, typically 100,000 observations, two separate statistical distributions are fitted to the resulting data set - a Gumbel distribution, and a bespoke distribution function known as the θ-function used to fit the observed maximum execution time. The Gumbel distribution provides an optimistic time-bound, whereas the θ-function offers a more pessimistic estimate. The combination of these two worst-case timing bounds are then used to provide an optimal execution time bound to schedule the real-time software tasks. Evaluating this approach, using a matrix multiplication application, and two sorting algorithms, the probability of exceedance for the timing bounds produced ranged between 10^{-4} to 10^{-6}.

Similarly, Hansen et al. [Hansen et al., 2009] present a statistical-based approach towards WCET estimation, by inferring the worst-case performance of an embedded software system using an estimation algorithm based on Extreme-Value Theory (EVT). The authors describe how they record over 200 million execution time measurements to form the basis for their later (off-line) statistical analysis. Rather than fitting a Gumbel statistical model to the entirety of the timing trace data, the authors explain how they group the timing measurements into variable sized blocks, selecting the maximum value from each block as the basis for later analysis. The measurements are further separated into estimation and validation sub-sets, to respectively fit and test the accuracy of the statistical model. The WCET bound produced using this method gives a probability of exceedance of 10^{-6}, or approximately one in a million, with any more accurate guarantees requiring a substantially larger set of timing measurements. However, the approach is applied only to statically-defined non-adaptable code, and evaluated off-line where there is no requirement to complete the statistical analysis quickly. The requirement for an exceptionally large set of timing measurements to generate an acceptable statistically-derived WCET bound precludes its usage within dynamically adaptable software systems. The limitations inherent in inferring worst-case behaviour from even an admittedly large set of representative timing measurements are apparent in the guarantees associated with the WCET bound. Any higher probability of exceedance would require an unfeasibly large set of timing trace data.

Analysis - Statistical techniques offer the greatest potential for predicting the likely execution time of dynamically-adaptable software. The various obstacles that make performing any exhaustive static analysis on dynamically-adaptable software impractical can be overcome using an empirical approach that infers future behavior from previous observations. However, since the performance of an adaptable system may vary, both between configurations, and within configurations, any statistical models of the execution time must be matched with a particular configuration of the system. As adaptations change the execution time behaviour of the system, the statistical models must be refreshed immediately after each adaptation, i.e., re-fitted using new timing data. The key problem with statistical modeling techniques, and the key challenge within this work, is how to make statistical inferences about the timeliness of the software, based on whatever limited knowledge might be available at adaptation-time. Both Hansen et al. [Hansen et al., 2009] and Edgar [Edgar, 2002], require a large number of timing measurements before a safe WCET bound can be produced. Collecting this data at run-time, in the context of a dynamically-adaptable system, would necessitate lengthy periods where no WCET bounds would be available.

The statistical methods outlined in this section are used to provide worst-case bounds for real-time systems, using the timing trace data gathered from trial runs as the basis for the statistical models. The WCET bound is produced off-line, however the timing data is gathered at run-time. The final statistical model fitting and WCET prediction is performed by a dedicated analyst, typically using a statistical package to evaluate the timing data.

2.3 Chapter Summary

This chapter has presented the state of the art in the timing analysis of static and dynamically adaptable software systems. A brief overview of the various adaptation frameworks was presented, ranging from off-line composition to dynamically-adaptable component-based systems. Adaptable systems, by their nature, alter their functional behaviour in unexpected ways at unanticipated periods during their execution, in order to better suit their operating environment. However, within closely-coupled software systems, modifying the functional behaviour of one part of the system can have repercussions on unadapted functionality elsewhere in the system. Functional adaptations may inadvertently alter the non-functional behaviour of the

Approach	Why?			Where?			What?		When?		Who?		How?					
	Real-Time Guarantees	Performance Analysis	QoS Maintenance	Source-code	Abstract Models	Timing Trace Data	ACET	WCET	Off-line	Run-time	Automatic	Supervised	Formal methods	System/State analysis	Timing Trace analysis	Flow analysis	Code Analysis/Simulation	Statistical Methods
Analysis Frameworks																		
Calinescu et al.			✓		✓	✓	✓			✓	✓			✓				
CiCUTS			✓	✓			✓	✓	✓		✓						✓	
Hissam et al.	✓					✓	✓	✓	✓			✓			✓			
KAMI			✓		✓	✓	✓			✓	✓			✓				
MESCAL	✓			✓				✓	✓			✓				✓	✓	
MOQA			✓		✓		✓		✓			✓		✓				✓
Formal Methods Analysis																		
AMEOBA-RT			✓		✓			✓		✓		✓	✓					
CLP	✓				✓			✓		✓		✓	✓					
Zhang et al.			✓		✓			✓		✓		✓	✓					
Measurement-driven Analysis																		
APS Analyzer	✓			✓		✓		✓	✓			✓			✓	✓		
MoDECS	✓			✓		✓		✓	✓			✓			✓	✓		
Tool-based Analysis																		
aiT	✓			✓				✓	✓			✓				✓	✓	
Chronos	✓	✓		✓				✓	✓			✓				✓	✓	
Heptane	✓			✓				✓	✓			✓					✓	
pWCET	✓			✓		✓		✓		✓		✓			✓	✓		
RapiTime	✓			✓		✓		✓		✓		✓			✓	✓		
SWEET	✓			✓				✓		✓		✓			✓	✓		
Statistical Analysis																		
Edgar	✓					✓		✓	✓			✓						✓
Hansen	✓					✓		✓	✓			✓						✓

Table 2.2: Summary of timing analysis approaches.

system, such as its execution time. Poorly understood timing behaviour hampers the ability of the software to function as expected, and can lead to missed deadlines, out-of-order execution or buffer overflows. The dependability of an adaptable system is degraded where uncertainties exist about its functional behaviour as well as its overall timing behaviour.

The principal focus of this chapter was an evaluation of the various timing analysis techniques currently in use, and their potential application towards dynamically-adaptable systems. The timing analysis methods were classified into static approaches, i.e., those applied off-line evaluating the software using a combination of code analysis and simulation, and dynamic approaches, i.e. those that rely on timing measurements to generate estimates of the likely future behaviour of the system. Current static timing analysis approaches are generally unsuitable for integration within dynamically-adaptable software, since they require lengthy off-line analysis periods, constrain the software, or demand extensive input from software developers and domain experts during analysis. Similarly, while the most sophisticated dynamic timing analysis methods can empirically evaluate software performance based on a series of timing measurements, they are not sufficiently reactive to estimate the changeable behaviour of dynamically-adaptable systems at run-time. Also, since the analytic process is typically supervised rather than automated, and is traditionally separated from the run-time environment, current timing analysis approaches are unsuited for unsupervised operation on resource-constrained embedded systems.

The next chapter describes a run-time reactive timing analysis approach, that can be integrated with dynamically-adaptable software running on a resource-constrained device. This approach provides a timing estimate immediately following an adaptation to the software, without having to halt the system to perform any analysis, or constrain the software to a set of pre-analysed configurations.

Chapter 3

TimePredict: A Reactive Run-time Timing Analysis

With four parameters I can fit an elephant,
and with five I can make him wiggle his trunk.
John Von Neumann

Chapter 1 introduced some of the basic concepts and motivations behind dynamically-adaptable software systems, and highlighted the challenges in predicting their execution time in the face of unanticipated functional modification. The previous chapter presented a review of current software adaptation frameworks, and the state of the art within software timing analysis methods. This chapter describes the design of TimePredict, a statistical-based timing analysis approach for dynamically-adaptable software systems. The primary goals of TimePredict are to accurately estimate both the worst-case and average-case execution time of run-time adaptable software, without halting the system or excessively impacting upon normal execution. TimePredict employs timing measurements to forecast software timeliness, and although several of the timing analysis approaches described in the previous chapter propose similar measurement-based analysis processes [Colmenares et al., 2008] [Wenzel et al., 2005], they each require extensive pre-generated timing information and perform the timing analysis exclusively from an off-line context. Simply applying these existing measurement-based timing analysis techniques to dynamically-adaptable

software would adversely affect normal execution, and restrict the scope and scheduling of adaptations to some previously observed configuration of the system. An approach is required that can predict the execution time of dynamically-adaptable software using run-time generated timing measurements, without restricting the scope of any functional adaptations or creating an excessive processing overhead on the system.

TimePredict forecasts the execution time of dynamically-adaptable software, by modelling software timeliness using statistical methods at run-time. The likely average-case and worst-case performance of the system can be inferred using these statistical models, and continuously updated with observations of the current timing behaviour of the software. Since the scope of the timing measurements available within a dynamically-adaptable system may be initially limited, i.e., immediately after an adaptation, TimePredict optimizes the predictive process depending on the timing information available, and the inherent execution-time volatility within the system. This approach enables accurate timing estimates to be generated for dynamically-adaptable systems, without halting the system to conduct the analysis or restricting the source, scope or scheduling of run-time functional adaptations.

The remainder of this chapter introduces the complexities implicit within software timing behaviour, and describes how the TimePredict approach forecasts the execution time of dynamically-adaptable software. The statistical models used within TimePredict are presented, and an outline of the model selection mechanism is provided, that determines the most appropriate predictive model given the current availability of timing measurements within the system. The process of continually updating these models, both during normal execution and at adaptation-time is also described. Lastly, this chapter outlines how TimePredict can offer statistically-derived timing estimates as feedback into the overall adaptation process.

3.1 Software Timeliness

Software timeliness can be defined as the period required to process a given task to completion, under typical operating conditions, on a specified hardware platform [Edgar, 2002]. A task that is repeatedly and regularly executed provides a convenient point of comparison to illustrate the potentially volatile timing behaviour of software executing within a live environment. Typically, the primary software control-loop provides the most representative measure of software timeliness, since it is executed continuously to achieve the principal goals of the system. However,

even relatively uncomplicated software may still exhibit highly variable timing behaviour between consecutive iterations of the same task, under similar operating conditions. Changing control flows, conditional branches, sub-loops, function calls or dynamic data structures may each create complex execution paths, and entail correspondingly complex software timing behaviour. In addition, fluctuations within the operating environment (e.g., due to user inputs or network connectivity issues) as well as changes within the system itself (e.g., processor load, execution history, memory usage or disc latency) may further alter software execution times in unexpected ways.

Figure 3.1 illustrates the type of highly variable timing behaviour that may be exposed through repeated measurement of a software task, based upon a similar example provided by Wilhelm et al. [Wilhelm et al., 2008]. The timing measurements can be summarized using a frequency distribution (histogram), in order to reveal the scale and variability inherent within the timing data. Timing bounds can be overlaid on this frequency distribution to describe the average-case and worst-case timing behaviour, as illustrated below.

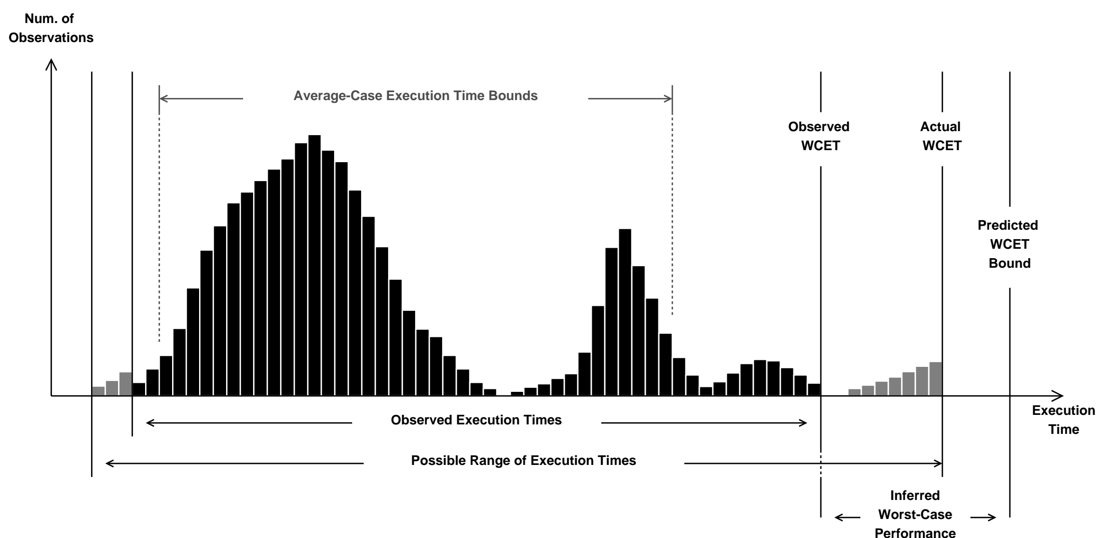


Fig. 3.1: Timing bounds used to define the execution-time performance of software.

Using a single term to characterize the overall timeliness of the software is insufficient, especially with complex execution time behaviour as shown in Figure 3.1. Software timing behaviour is more readily described in terms of whether it represents an average-case or best/worst-case execution time. The former provides an indication of the central tendency of the software perfor-

mance, whereas the latter is used to describe the extreme range of potential timing measurements capable of being produced by the software. Traditionally, timing analysis techniques have been concerned with deriving worst-case bounds to ensure the safe scheduling of tasks within hard real-time and embedded systems [Colin and Puaut, 2001] [Ermedahl et al., 2005]. However, the worst-case execution time is a rarely (if ever) occurring event that fails to represent the more usual execution-time performance of the software. A bounded average-case execution time, capturing a specified percentage of the execution time measurements of the software, provides a more directly accessible metric in describing the typical software timeliness [Schellenkens, 2010]. The various characteristics of software timeliness illustrated in Figure 3.1 can be defined as follows;

- Average-Case Execution Time (ACET) Bounds,
  The ACET uses an upper and lower bound to encompass a specified percentage of execution-time measurements of the software. The percentage selected is typically between 50% and 100%, and is analogous to the inter-quartile or inter-decile range used to measure the statistical dispersion of the timing measurements (see Glossary).

- Observed Worst-Case Execution Time (WCET),
  The observed WCET is the largest value within the set of existing software execution time measurements. Typically, the more timing measurements that are made of the system, the closer the observed WCET comes to the actual WCET [Hansen et al., 2009].

- Actual WCET,
  The actual WCET is the maximum theoretic worst-case value that may occur during the execution of the software. This single extreme execution time may not be exposed during run-time testing or execution, but must generally be inferred from the existing data using a pessimistic timing analysis [Wenzel et al., 2005].

- Predicted WCET,
  The predicted WCET bound is a pessimistic timing bound, created through the analysis of existing timing measurements. The WCET bound is pessimistic in that it over-estimates the worst-case performance of the software, in order to safely encapsulate its extreme timing behaviour, i.e., the actual WCET value.

Estimating the average-case and worst-case execution times of a dynamically-adaptable software system is the subject of the TimePredict approach introduced in this chapter. Since

the functionality as well as the timing behaviour of the software can change during run-time, TimePredict continuously forecasts the likely execution time of the current configuration of the software. Any run-time functional adaptations to the software are detected by TimePredict, and result in corresponding adjustments to the timing estimates for the new configuration of the system. Within Figure 3.1, the observed and actual WCET values are presented separately, since a measurement-based approach such as TimePredict may not expose the extreme timing performance of the software directly through empirical testing. For example, the WCET may not be directly observed even after repeated measurements of the software executing under a variety of operating conditions, inputs and load levels [Colmenares et al., 2008]. Consequently, predicting the likely WCET value may over-estimate the actual (unobserved) WCET performance of the software, in order to provide a greater confidence that the absolute worst-case timing behaviour has been encapsulated within the WCET bound.

In contrast, the ACET can be defined, using an upper and a lower bound, as a range of timing values within which a specified percentage of the software timing measurements are expected to be found. This percentage, typically describing a majority of the existing timing measurements (i.e., $\geq 50\%$), allows for a more qualitative comparison of the execution time performance of the software. Using a single metric, such as the mean timing value, is impractical since it provides a poor estimate of central tendency. While the WCET should not greatly change during run-time, the ACET may vary considerably, depending on the current operating conditions and changing load on the system. An accurate estimate of the likely ACET of the system provides a valuable insight into the effectiveness of the software at accomplishing its principal goal. Since changes within the operating environment may be exposed more readily through estimating the average-case performance of the software, predicting the ongoing ACET can provide an early indication of when adaptations may be required.

Both ACET and WCET estimates have associated levels of accuracy and precision, the former indicating how often in percentage terms the provided timing bounds actually encapsulate the next timing measurement, and the latter providing the average time difference in milliseconds between the timing bounds and the next timing measurement. An idealized predictive process would have an accuracy of 100%, and a precision of 0 milliseconds, however in practice this level of performance is difficult to achieve. Typically, there exists an inverse relationship between accuracy and precision, where improving one degrades the other, e.g., a WCET bound set at an

extremely large value may have close to 100% accuracy, but poor overall precision.

	Average-case accuracy	Worst-case accuracy	Average-case precision	Worst-case precision
Off-Line Timing Measurements				
Off-line timing data taken using different hardware	N/a	N/a	N/a	N/a
Off-line timing data taken from comparable hardware	L	L	L	L
Off-line timing data taken from the same hardware/system	M	M	M	M
Run-Time Timing Measurements				
No run-time or off-line timing data available initially	N/a	N/a	N/a	N/a
Few run-time measurements available ($n < 30$)	L	L	L	L
Some run-time measurements available ($30 < n < 100$)	M	L	M	L
Many run-time measurements available ($100 < n < 5000$)	H	M	H	M
Expansive run-time measurements available ($n > 5000$)	H	H	H	H

Table 3.1: Effect of measurement availability on timing estimates.

Table 3.1 describes the expected accuracy and precision of TimePredict's predictive process under a number of different assumptions. Both the accuracy and precision of timing estimates can be expected to be low (L), medium (M) or high (H), or potentially unavailable (N/a) as the circumstances changes. In practice, the number of available timing measurements, as well as how closely these measurements represent the actual timing behaviour of the system, will determine the accuracy and precision of the predictive process. The number of measurements required for a low, medium or high degrees of accuracy and precision are illustrated in Table 3.1 using somewhat arbitrary values ranging from less than 30 to over $5,000$. These figures are only illustrative, since highly variable software timing behaviour may require significantly more measurements than more stable software configurations, to generate timing estimates to the same level of accuracy and precision.

Within systems that can be evaluated off-line, a large number of timing measurements can be

generated for later analysis, since the testing period does not coincide with normal execution on a live system. However, if these off-line timing measurements have been generated on a different hardware environment, or under different operating conditions, the measurements may not be truly representative of the timeliness of the system once it is deployed within a live operating environment. While run-time generated timing measurements provide the best indication of the actual behaviour of the system, they must be generated concurrently with the execution of the software. Unfortunately, this may lead to periods, e.g., immediately following an adaptation, where only limited timing information is available to TimePredict, but timing estimates must still be produced.

The TimePredict approach, described in Section 3.2, provides a reactive timing analysis process capable of contending with different operational scenarios, so that a timing estimate can be produced even when limited timing information is available. Within dynamically-adaptable systems, composed of shifting functional elements, the timing behaviour of the software is never fixed, but may change at any period with little prior notification. TimePredict must allow timing estimates to be refreshed at run-time, to match functional changes to the underlying software. Using a measurement-based approach, TimePredict ensures that timing estimates correspond to the actual performance of the software, rather than any theoretic analysis of the system under idealized conditions. Section 3.2 describes the run-time timing analysis process and introduces the statistical models used to derive the ACET and WCET timing estimates. However, before this predictive process is presented, we must first describe the constraints imposed on TimePredict by the nature of its operating environment.

3.1.1 Operational restrictions

TimePredict is designed to forecast the execution time of software running within a resource-constrained operating environment, e.g., within an embedded system containing no more than several megabytes of memory, and a single MHz-scale processor. The predictive process must be carefully regulated to avoid placing excessive demands on the limited resources of the underlying system, or disrupting the normal behaviour and performance of the software. Similarly, the estimates themselves need to be generated quickly at run-time, while the processor is busy performing other tasks. Consequently, the forecasting method employed by TimePredict is both a product of its operating environment, and the requirements for accuracy and precision within

its timing estimates.

There are three primary challenges to overcome in designing a run-time predictive process suitable for use within embedded devices, namely, to minimize use of the meager system resources, to maximize the accuracy of the predictive process with the data available at run-time, and to generate estimates at run-time without negatively impacting on performance. Failing to overcome any one of these challenges can fatally compromise the predictive approach as a whole. For example, a highly-accurate, but resource-intensive, forecasting method can consume processor or memory resources to the detriment of other system tasks. Likewise, an efficient, but inaccurate predictive process may produce estimates that provide no actionable information about the current status of the system.

Although the TimePredict approach is designed specifically to estimate the timeliness of resource-constrained embedded devices, the predictive models used could be applied to any number of other processes or system behaviours. Other non-functional behaviours within the system, such as the network bandwidth, task throughput or power consumption, may exhibit similar non-deterministic behaviour, and could be similarly measured and predicted at run-time. Alternatively, other processes outside of the area of embedded systems could likewise be used as the basis for run-time estimates, e.g., weather forecasting [Cadenas and Rivera, 2010], predicting the volatility of financial options [Chou, 2005], estimating financial risk [Rosenberg and Schuermann, 2006] or as a means of predicting the rate of inflation [Engle, 1982]. Both software timing analysis and other complex systems share similar properties, including markedly distinct periods of volatility, increased uncertainty in the presence of few reliable measurements, and a complex cause-and-effect relationship between the process being measured and its environment.

The TimePredict approach, while being a product of its target operating environment, can potentially be applied within other situations to predict the performance of other processes. Indeed, since the predictive models employed by TimePredict are measurement-based, data from any other run-time process or time-series could be substituted for the software timing measurements, without any further alterations being required. While the application of TimePredict to other problems is outside the scope of this thesis, it does form a potential area of future work.

3.2 The TimePredict Approach

The TimePredict approach provides a reactive run-time statistical-based timing analysis method, to forecast the worst-case and average-case execution times of dynamically-adaptable software. The principal focus of this thesis, and the aim of TimePredict, is on predicting the timeliness of dynamically-adaptable software, and does not consider in detail any issues related to adaptation selection, run-time software optimization, or functional analysis within adaptable systems. However, for the purposes of demonstrating the TimePredict approach, an implementation of a dynamically-adaptable system is required containing some of these features. Figure 3.2 illustrates how TimePredict integrates within the architecture of a dynamically-adaptable system, measuring the ongoing execution time of the current configuration of the software, and generating ACET/WCET estimates that can provide feedback into the adaptation selection process.

The scope of this thesis is delineated solely by the TimePredict approach, so that the implementation of the Adaptation Manager, Adaptation Engine, Component Repository and Component Model are used to demonstrate the approach, but are not individual contributions of this thesis in themselves. A more detailed description of these components, including the selection and scheduling of adaptations within a dynamically-adaptable system, is provided by Fritsch and Clarke [Fritsch and Clarke, 2008]. The various actions performed by the system, as illustrated within Figure 3.2 and labelled 1 to 5, can be described as follows;

1. The execution time of the primary control loop within the component assembly is measured, and the timing observation recorded by TimePredict.

2. TimePredict analyses the timing data, and updates its existing set of timing measurements if necessary (see Section 3.2.2). This updated timing information is then used to refresh the ACET/WCET timing estimates. These estimates are forwarded to the Adaptation Manager for additional analysis.

3. Within the Adaptation Manager, the rules engine evaluates the current ACET/WCET estimates and determines whether an adaptation is required. Should an adaptation be deemed necessary, the decision engine selects the most appropriate change to make to the software. This adaptation plan is then forwarded to an Adaptation Engine for implementation.

4. The Adaptation Engine determines the scope and scheduling of an adaptation, based on the

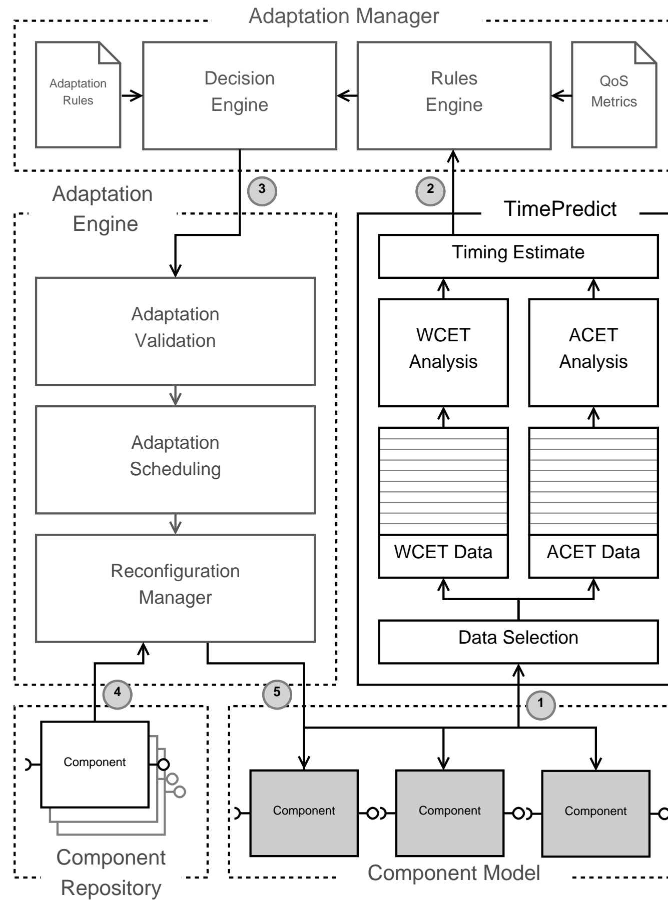


Fig. 3.2: Architecture of a dynamically-adaptable system featuring TimePredict.

adaptation plan created by the Adaptation Manager, as well as the availability of alternate software components within the Component Repository.

5. The reconfiguration manager within the Adaptation Engine adds, replaces or removes target components from the Component Model to implement the adaptation. This process is performed at run-time, resulting in a new configuration of software components with optimized timing behaviour, based on the original timing feedback generated by TimePredict.

Dynamically-adaptable systems that support unrestricted functional adaptation can modify their behaviour, at run-time, in a manner that may not have been foreseen by the original software developer. Since the scope of an adaptation cannot be restricted to a set of pre-defined

configurations of the software, the performance of the system cannot be exhaustively tested from an off-line context. Similarly, since the selection and scheduling of adaptation actions are performed at run-time, very little opportunity exists to perform pre-emptive run-time analysis on the next potential configuration of the system, i.e., the time it takes to initiate and implement an adaptation largely precludes a thorough adaptation-time software timing analysis. In order to establish the timeliness of an unrestricted dynamically-adaptable system, a concurrent run-time timing analysis process is required.

A run-time timing analysis process must both generate measurements of the underlying system, as well as predict its likely future timing behaviour. The timing measurements themselves are generated within the Component Model, by measuring the primary control loop of the current configuration of the software (see 1 in Figure 3.2). Measurements are forwarded to TimePredict and selected for either worst-case or average-case analysis (or both), since the former is more concerned with extreme timing values, while the latter is solely intent on recent timing behaviour. The data is stored within TimePredict and an analysis performed, resulting in a combined ACET/WCET timing estimate. Since TimePredict is a measurement-based approach, the predictive process used is directly related to the number of measurements available within the system. When a large number of timing measurements have been recorded, TimePredict provides a high-confidence timing estimate to the Adaptation Manager, i.e., an ACET/WCET estimate where the probability of encapsulating the next timing measurement is likely to be between 95% and 99.99999%. Although the definition of what probability forms a high-confidence estimate is somewhat arbitrary, depending on the nature of the software and its operating environment, the latter probability is stated by Hansen et al. [Hansen et al., 2009] as defining a high-confidence estimate within embedded software systems. Within hard real-time systems however, this value may rise to probabilities of exceedance on the order of 10^{-12} [Bernat et al., 2003]. For the purposes of unrestricted dynamically-adaptable systems, the level of confidence in the estimate will grow, from an initial speculative forecast when timing information is limited, to a high-confidence estimate above 95% when many timing measurements are available.

The expected functional and non-functional behaviour for the system as a whole (or a configuration thereof) is recorded as a series of Quality of Service (QoS) requirements, and processed using the Rules Engine (2). In effect, the timing estimate provided by TimePredict allows the QoS obligations to be evaluated, and if a performance deficiency is discovered, an adaptation can

be quickly initiated. However, adaptations must be constrained by the availability of alternate software behaviours, the nature of the performance deficit and scheduling issues surrounding any proposed adaptation action(s). To enact a desired adaptation, the Decision Engine (3) consults with the Component Repository (4) to schedule and implement the desired change (5). The newly adapted software configuration then re-starts execution, and provides a fresh set of timing measurements to re-fit the timing models within TimePredict (1). This feedback cycle of measurement, timing estimation and functional adaptation facilitates the on-going optimization of the system, in response to changes in the operating environment.

The following sections describe in more detail the design challenges faced within the TimePredict approach, with respect to the measurement of software execution times, the selection of appropriate timing data, and the analysis of this data to form an estimate of the expected future performance of the system.

3.2.1 Timing Measurement

TimePredict uses representative timing measurements, taken from the current configuration of a dynamically-adaptable system, to generate accurate worst-case and average-case timing estimates. Since the functional composition of dynamically-adaptable software is liable to change unexpectedly at run-time, a common feature within each configuration of the software is required to provide a comparative timing measurement both within and between configurations. Although the TimePredict approach is functionally agnostic, i.e., the execution time of any part of the system can provide the basis for the predictive process, a regularly executed software function offers the most meaningful point of comparison and analysis - both within and between the various configurations of a dynamically-adaptable system.

A common software task or function, or even different functions undertaking similar logical roles within the system, can be repeatedly measured to provide an on-going snapshot of the overall performance of the system. In the case of a closed-loop system, its primary control loop should provide a convenient point of comparison and measurement. Typically, closed-loop control systems rely on context information about their operating environment (e.g., sensor data), to provide on-going feedback into a continuously executing control function. This function then evaluates the current context information and determines whether any changes to the system are required to ensure optimal performance, e.g., through adjustments to any attached motors,

actuators or operating parameters [Cervin et al., 2003]. Since the control function is executed time after time, repeatedly measuring its performance can provide a representative measure of the execution time of the system as a whole. Dynamically-adaptable systems using this type closed-loop control, can initiate adaptations when the operating environment changes sufficiently so that the range of functional behaviours expressed in the control function are no longer optimal for the prevailing operating conditions. Even though the functionality within the control loop can be altered through run-time software adaptation, its role within the system, and its usefulness as a point of timing measurement, remains the same.

In the case of event-driven systems, where no single main control loop exists, software tasks can be executed to perform the same evaluation and ongoing adjustment to the system. However, unlike closed-loop systems, the software tasks within event-driven systems may either be scheduled for repeated execution at regular intervals, or executed only intermittently, at indeterminate periods in response to either internal or external events. A regularly executed software task provides an excellent point of measurement for the system, assuming it is important enough to be included within each configuration of a dynamically-adaptable system. A purely event-driven software task, may perform a valuable service as a point of measurement, if the events triggering its execution are somewhat regular, its usage is common across all configurations of the system, and its operation is sufficiently complex to serve as a representative measure of the overall system performance. The timing behaviour of event-driven systems may be further complicated if software tasks can be preempted (paused and restarted) during execution. Choosing a software task to form the basis for the timing analysis process must first ensure that task preemption is avoided wherever possible, however estimating the timing behaviour of tasks within preemptive systems can be successfully achieved [Ghosal et al., 2004]. If a software task must be selected for timing analysis, its operation must be relatively important to the operation of the system, its execution reasonably regular, and its priority/scheduling sufficient so that disruptions due to run-time preemption are minimized. Within dynamically-adaptable event-driven systems, some tasks should be common across each configuration of the software, and executed with enough frequency to provide the basis for a timing analysis.

All measurement-based timing analysis approaches rely on accurate observations of the software timeliness, using a process that avoids any unwanted interference with the normal execution of the system. Within both C/C++ and Java, millisecond-accurate timing measurements are

readily available [Corsaro and Schmidt, 2002], and allow the elapsed time to be calculated by finding the difference between two consecutive timestamps. More fine-grained timestamps are possible using more advanced techniques, e.g., such as counting the number of processor cycles (see Section 3.2.1.1), however millisecond-accurate timing measurements are more common, and accurate enough for the type of soft real-time applications considered within this thesis. However, the TimePredict approach can be equally applied to any clock resolution without any alterations to the predictive models, since the analysis process is based purely on the numeric values of the software timing measurements.

The timing estimates produced by TimePredict typically focus on predicting the period of time required for the flow of execution to enter/exit a representative software function, such as the primary control loop [Westermann and Happe, 2010], or some other characteristic software task within the system [Pop et al., 2008]. The function or task that is selected as the basis for the timing analysis, is chosen due to its performance being fundamental to the functional behaviour of the system as a whole, and therefore a good indicator of the overall system timeliness. A primary control loop provides the most representative measure of software timeliness, since it is executed continuously, required within each configuration of the system and allows inferences to be made by comparing different timing measurements of the same functional cycle. The code is instrumented with timestamps to the measure the start and end of the function, and the observed execution time then forwarded to TimePredict . In practice, this type of measurement process adds very little additional processing overhead to the software, and any requirement to store the timing measurements for analysis is offset by the limited subset of data required by TimePredict to generate timing estimates (see Section 3.2.2). However, every measurement process is susceptible to error, and millisecond-accurate software timestamps are no different. Since the clock resolution of the system will remain fixed during execution, this error should constitute at most a very small percentage value within each timing measurement, as described in the next section.

3.2.1.1 Clock Resolution

The system clock (also occasionally referred to as a software clock) provides embedded and desktop processor architectures with a readily accessible means of measuring elapsed time, including the execution time of software. The use of expensive timing hardware is impractical in cheap

embedded computing devices, both from the stand-point of cost as well as power consumption. Typically, quartz crystals are employed as an oscillator, since they can be manufactured cheaply and easily to provide an alternating voltage at a known frequency. This alternating voltage provides the basis for the system clock, allowing specific periods of time to be measured out using dedicated counters within the hardware [Pásztor and Veitch, 2002]. When the counter reaches a certain value, an interrupt is generated to provide the operating system with a standard repeating time signal. This time signal, rather than the more fine-grained oscillations of the crystal, is available via the operating system to measure the timeliness of software. Each operating system defines a minimum measurable time period, known as the clock resolution, that varies from system to system. Typically most Linux-based operating systems provide a 1 millisecond (ms) clock resolution, although specialized hardware or O/S modifications can offer a more fine grained clock-resolution if required [Corsaro and Schmidt, 2002].

A default clock resolution of 1ms implies that time periods on the order of several milliseconds are susceptible to significant measurement error. For example, if the 'true' execution time of a software task ranged from 2.5ms to 5.5ms, the rounding error associated with the default 1ms clock resolution would greatly affect the accuracy of any timing measurements. For the purposes of this thesis, it is assumed the typical software execution times are on the order of several tens or hundreds of milliseconds, so that the measurement error is reduced to a negligible percentage within each timing measurement.

3.2.1.2 Clock Drift

While the measurement error must be minimized within individual timing measurements, the inherent error within the system clock as compared to another time source is less significant. If timing measurements are carried out exclusively using the on-board system clock, its inherent error will be imparted to all timing measurements equally. However, the magnitude of this error may change over time when manufacturing errors, temperature changes, or even particle radiation causes the quartz crystal to oscillate at slightly longer or shorter intervals [Schmid et al., 2008]. This causes the clock to fall out of sync with another (previously synchronized) clock source, resulting in a condition known as clock drift.

Timing errors due to clock drift may be corrected in software, by periodically accessing a network-time server [Mills, 1990], to re-synchronized the system clock. This type of distributed

clock synchronization approach can bring two clock sources back into agreement, to an accuracy of approximately of 1ms [Corsaro and Schmidt, 2002]. However, the TimePredict approach assumes that all timing measurements are generated exclusively within a single hardware environment, using a single on-board clock source. Clock drift is unlikely to become apparent unless the timing measurements being generated by the system are on the order of hours or days in duration [Corsaro and Schmidt, 2002], rather than several hundred milliseconds. This enables safe comparisons between individual timing measurements, even though the clock source may be in error compared to the 'true' time. If required, maintaining a synchronized system clock, e.g., for correct time/date information, can be achieved by periodically synchronizing the system clock at adaptation-time, to correct any accrued errors caused by clock drift.

3.2.2 Data Selection

Storing every timing measurement generated within the system is not only potentially costly in terms of the system resources consumed, such as memory and persistent storage, but is wholly unnecessary once a sufficient number of representative timing measurements have been recorded. The obvious limitations of resource-constrained embedded devices encourage an optimal use to be made of the minimal resources available within the system. The average-case timing analysis process employed by TimePredict (described in Section 3.3), places a greater predictive value on more recent timing measurements. To limit the amount of extraneous timing data, only a specified number of timing measurements are stored at any given time. The size of this data set should be sufficient to perform a detailed timing analysis, but not needlessly overburden the system. Since timing measurements are constantly generated within the system, the data set is regularly refreshed with more recent measurements, and older measurements discarded. A First-In First-Out (FIFO) array, of a specified size, is used for this purpose. The array is filled with timing measurements in order of their occurrence, and once filled, the oldest value in the array is over-written with the latest timing measurement.

In contrast, worst-case timing analysis highlights the magnitude of timing measurements over the order of their occurrence. TimePredict uses a minimum-value ranked array to record extreme timing measurements for worst-case analysis. Timing measurements are added to the array in ascending order, using an in-place sort, until the array is filled. Subsequent timing measurements are only added to the array if they exceed a minimum value. The array itself is

used to construct a frequency distribution (histogram) of the timing measurements, effectively counting the number of occurrences of a particular value. Any timing measurements that surpass the current observed maximum (worst-case) timing measurement, result in the array being re-ordered, essentially replacing smaller ranked values with the new maximum timing measurement. Initially, when few timing measurements are available, the worst-case array will be periodically re-sorted, to include more extreme timing measurements, however, after the likely worst-case behaviour has been established, changes to the array will typically take the form of adding a new measurement to the existing frequency distribution. Since the worst-case analysis is focused on extreme, rather than average timing measurements, any values smaller than the current minimum value within the array are ignored, as they offer no further insight into the worst-case timing behaviour of the system. All the array values maintained by TimePredict contribute to the analysis of the timing behaviour of the system, thereby conserving system resources from being needlessly squandered maintaining irrelevant data.

Both the average-case and worst-case FIFO arrays can be set to an arbitrary length, and store an arbitrary number of timing measurements for analysis, however, the probability of exceedance for either the ACET or WCET bounds will not be more than 10^{-5}. Depending on the inherent variability within the timing behaviour of the software, a sample size of $5,000$ measurements would be the upper range required to satisfy this level of confidence in the estimate. In practice, 50 or 100 measurements should be sufficient to maintain a representative sample size at any one time, and entail only very modest demands on either the processor or system memory. The next chapter introduces in greater detail the demands imposed on the system memory by these data structures, typically, within 32-bit systems using integer arrays, each array element consumes 4 bytes of memory space [Arnold et al., 2005]. At the extreme case for TimePredict, two $5,000$ element arrays containing the ACET and WCET-specific data would therefore consume approximately 40kB within memory.

The on-going analysis process within TimePredict continuously updates these arrays with the latest timing measurements. This ensures that any unexpected timing perturbations, caused by changes in the operating environment, are immediately registered and the appropriate adjustments made to the timing estimates. However, functional adaptations alter the timing behaviour of the software, and invalidate any previously recorded timing data. To make sure that no invalid timing measurements are maintained, any functional adaptations to the system are preceded by

a signal (from the Adaptation Manager) to TimePredict, indicating that all the ACET and WCET array elements can be removed, to be replaced by a more representative set of timing measurements.

3.3 Average-Case Analysis

While the worst-case timing estimates provide an insight into extreme, rarely-observed timing behaviour, the average-case execution time (ACET) can provide a more informative metric to describe the typical performance of the software. Correctly estimating the ACET provides a short-term estimate (to a high level of confidence) of software performance under the current operating conditions. Rather than comprising a single value, the ACET is described in terms of a bounded timing estimate, with an upper and lower bound that encapsulates the expected timing behaviour of the software (as outlined previously in Figure 3.1).

TimePredict estimates the ACET of the software using a run-time measurement-based approach. The principal drawback of using timing measurements within unrestricted dynamically-adaptable systems is that newly adapted configurations of the software may not have been previously encountered, nor any timing measurements collected in advance of execution. Immediately following an adaptation, there may exist a limited amount of timing data, making any subsequent measurement-based timing analysis process difficult. Many of the statistical-based timing analysis methods commonly applied to predict software ACET or WCET usually require a large number of timing measurements, occasionally in excess of a million values [Hansen et al., 2009], to generate a safe, accurate timing estimate with a probability of exceedance far greater than 10^{-5}. Unfortunately, within dynamically-adaptable systems, the set of timing measurements that form the basis for the ACET estimate must be generated at run-time, thereby denying any opportunity to perform any detailed statistical analysis involving large datasets.

Since the TimePredict approach may have to forecast the execution time of software using a restricted set of timing measurements, two separate predictive processes are used in the analysis of worst-case execution times, one to accommodate a reduced set of timing measurements, and a second when sufficient data has been collected at run-time. This is due to the worst-case behaviour of the system not becoming immediately apparent over the course of a few timing measurements (i.e., there may be a significant difference between the observed worst-case and actual worst-case timing behaviour). The average performance of the system however can be

assessed using relatively fewer timing measurements, since the typically timing behaviour can be inferred using every measurement generated within the system. TimePredict uses an Exponential Smoothing (ES) model to predict average-case timing behaviour, since it can provide accurate timing estimates with a limited set of timing measurements. The ES model parameters are adjusted at run-time as more measurement data becomes available, but there is no switch-over between different models as is the case with the WCET analysis. The next section describes the Exponential Smoothing model, and its application at run-time to generate the ACET estimates for dynamically-adaptable software.

3.3.1 Exponential Smoothing Model

TimePredict uses an Exponential Smoothing (ES) model, to predict the average-case performance of the software using a limited number of timing measurements available within the system. The ES model evaluates recent timing measurements, as well as the previous ES estimate, in order to forecast the likely execution-time behaviour of the system. Since only a limited number of terms are required to generate a timing estimate, the ES model provides a convenient predictive process immediately following an adaptation, when few timing measurements have been collected. As more measurements are gathered, the ES model may become less reactive to sudden changes in timing behaviour, and provide a stronger emphasis on longer-term trends within the data.

The ES model produces estimates using a relatively small set of measurements, and consequently has a smaller computational cost and memory overhead compared to other statistical modelling techniques (including the GEV model used for worst-case timing analysis). This allows the ES model to be continuously updated during run-time without excessively impacting on the underlying system. By including the previous estimate, as well as the most recent measurement as distinct terms, the ES model can balance the effects of short-term variation and longer-term trends within the timing behaviour of dynamically-adaptable software. This balance, achieved through adjusting the smoothing parameters associated with each term, can be used to 'reset' the model after an adaptation occurs. For example, sudden changes in the measured timing behaviour of software can be quickly incorporated within the ES estimate, with the effects of this changes fading uniformly as variation decreases.
The ES model starts by setting the initial estimate to the first timing measurement (t_0), i.e.,

$$S_{n+1} = \begin{cases} t_0 & \text{if } n = 0, \\ \alpha t_n + (1-\alpha)S_n & \text{if } n > 0. \end{cases} \tag{3.1}$$

The basic ES model described in Equation 3.1 provides a point estimate of ACET behaviour, i.e., it forecasts the next measurement in the time series describing the execution-time performance of the software. A time series is used to describe software timeliness, since the occurrence of each timing measurement is significant. It is worth noting that the term time series, has no relation to software execution time, but refers to a sequence of measurements recorded in the order of this occurrence at successive time intervals. Examples of time-series data include rainfall patterns and temperatures, solar sunspot activity, commodity prices and airline passenger numbers [Chatfield, 2003]. Within each of these data sets, subtle correlations can be seen within forming patterns the data, e.g., typically daily temperature measurements are higher in summer than in winter. Similarly, with execution time performance within embedded systems, if the workload is light and the system resources are available, the execution time may be shorter than an over-burden system with too few processing resources available.

Exponential smoothing models are so called since they include a smoothing factor, in this case α, to provide a bias either towards more recent timing measurements or emphasize previously generated ES estimates (longer-term trends within the observed timing behaviour). The smoothing factor is typically set to some value in the range 0.0 to 1.0 [Gardner, 1985], however, the value itself must be estimated during run-time to provide at appropriate fit to the data. Although the exponential smoothing model described in Equation 3.1 estimates the ACET, it does not incorporate any apparent trends in the data into its estimate, nor does it provide any bounds on the estimate itself. By expanding the basic ES model to include an error parameter, we can provide a bounded ACET estimate rather than a single point estimate. A bounded estimate may be more instructive in the case of average timing behaviour, since it allows the typical range of values to be specified. This updated smoothing model, presented in Equation 3.2 begins in a similar fashion to the basic ES model (Equation 3.1), by setting the first timing measurement (t_0) as the initial timing estimate. Once three or more timing measurements have been collected, a bounded estimate can then be produced, derived from both the point estimate (S_n) and error term (ϵ_n), i.e.,

$$X_{n+1} = \begin{cases} t_0 & \text{if } n = 1, \\ \alpha t_n + (1-\alpha)X_n & \text{if } n < 2, \\ S_n \pm \epsilon_n & \text{if } n \geq 2. \end{cases} \tag{3.2}$$

X_{n+1} provides the bounded ACET estimate, S_n defines the previous timing estimate and ϵ_n describes the error term used to construct the estimate bounds. The parameter n defines the number of measurements that have been recorded within the system, while the smoothing parameter α is the same as that described in Equation 3.1. The two equations used to estimate the timing and error values can be defined as,

$$S_n = \alpha t_{n-1} + (1-\alpha)X_{n-1} \tag{3.3}$$

$$\epsilon_n = \beta(X_{n-1} - t_{n-1}) + (1-\beta)\epsilon_{n-1} \tag{3.4}$$

Two smoothing factors are included in the above equations, α associated with the timing data and β associated with the errors in the estimates, and are both similarly defined within the range of 0.0 to 1.0. Gardner describes a method of estimating these smoothing parameters using an iterative approach based on the data [Gardner, 1985]. In essence, the smoothing parameters can be derived by converging on values that minimize the mean squared error of the estimate, i.e., the sum of the squared differences between the previous estimates and the subsequent timing measurement. Starting with an initial range for α and β, and iterating through this range in fixed increments, a good fit for both smoothing parameters can be quickly found. The algorithm used to estimate the smoothing factors for both the timing and error terms is described in Equation 3.5.

$$(\alpha, \beta) = min\left(\sum_{i=0}^{n}(t_i - X_i)^2\right) \qquad \forall\alpha \in [0,1], \forall\beta \in [0,1] \tag{3.5}$$

The mean square error (MSE) finds the difference between the previous measurements (t_i), and their respective estimates (X_i) for changing values of α and β. The minimum MSE indicates the best fit for the two smoothing factors when applied to the existing data. The computational

costs of this parameter-fitting method can be increased or decreased by changing the size of increment used to iterate through the allowed ranges for α and β. By default, the algorithm to find the best fitting estimates for α and β is called recursively, using ever smaller increments, until an increment size of a particular threshold is reached. By default, TimePredict has set this threshold so that it finds approximations for α and β accurate to $\pm$ 0.001. This limit was chosen since if provided the best accuracy without placing excessive demands on the available memory and processor resources. More accurate approximations are possible, given greater time and more resources invested in their calculation, however, the likely increasing in predictive accuracy will be small as the threshold decreases further.

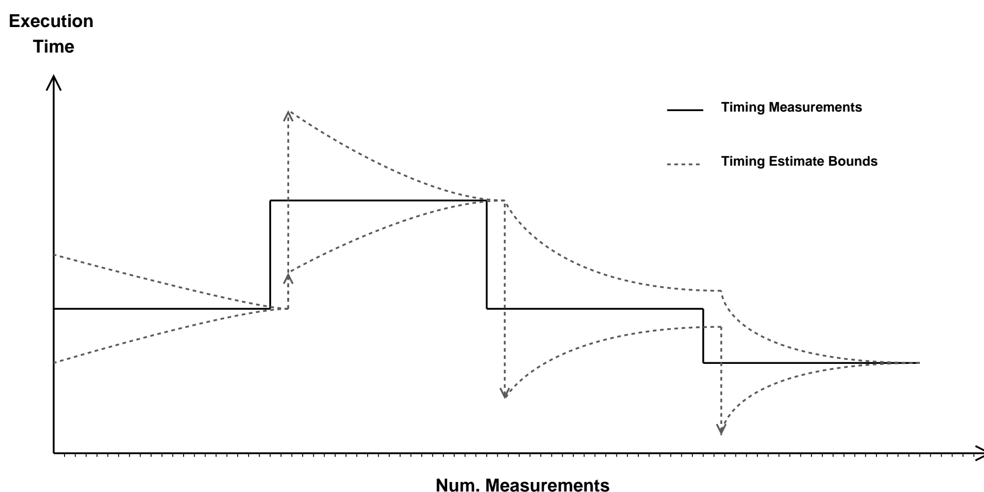


Fig. 3.3: Bounded timing estimates using the exponential smoothing model.

Unlike the timing estimate outlined in Equation 3.3 the error term (ϵ_n) is calculated separately to provide an upper and lower timing bound. The error term governing the upper and lower case bounds (3.4) will grow in the case where either the previous timing estimate bounds are exceeded, or when the variation within the timing measurements increases. Otherwise, the error term will decrease at a rate determined by the smoothing factor β, again set to some value between 0.0 and 1.0. Similarly, the determination of β is made at run-time, by iterating over the previous errors for the current value of α. Similarly the previous error (ϵ_{n-1}) is used provide an added bound if there is a trend apparent within the error terms, and allows the ES model bounds to roughly follow linear trends within the data. The impact of the error term on the overall timing estimate bounds can be illustrated in Figure 3.3. This figure shows the type of bounded

timing estimates produced using the exponential smoothing model. The error associated with each timing estimate reduces as the timing measurements stabilize at a particular value, i.e., the error term tends towards zero as the timing behaviour becomes more stable. This is illustrated in Figure 3.3 as both the upper and lower estimate bounds converge on the timing measurement if it remains unchanged over a given period. When the timing measurement changes unexpectedly, the bounded timing estimates increase to capture the new (and unanticipated) timing behaviour. These new estimate bounds are re-fixed at a higher or lower value, and begin again to converge if the timing measurement values remain relatively static.

While this ES model provides a convenient method of forecasting the execution time of software, the model does not perform any historic analysis of previously collected data. The ES model operates at run-time on a per-measurement basis, and therefore is limited in its predictive abilities compared to a statistical approach using a sufficiently large set of timing measurements. However, the advantages of the ES model lie in its ability to both provide timing estimates based on very limited information, and by being able to quickly react to small variations in the software timing behaviour.

3.3.1.1 Critique of the ES Model

The ES model, as included within TimePredict, provides a number of clear benefits when estimating software timeliness within restricted operating environments. This includes reducing any negative impact on the performance of the underlying system, as well as the predictive advantages offered by the use of the ES approach itself. The benefits associated with the ES model include;

1. Reduced memory usage.
2. Low processor overhead.
3. Fast execution.
4. Good accuracy and precision.
5. Ability to generate estimates based on a small sample size.
6. Reactive to sudden changes in the underlying process (i.e., changes in software timeliness).

Although there are potentially more accurate ACET estimation methods [Schellenkens, 2010] [Guo et al., 2008], none can derived estimates at run-time, let alone execute within a resource-constrained operating environment. The use of the ES model within TimePredict was carefully considered to maximize predictive performance while simultaneously minimizing resource usage and avoiding any negative impact on the underlying system. In addition, since the execution time behaviour of the software can change quite suddenly (due to adaptations), any ACET analysis method used within TimePredict must quickly reflect these timing changes within its estimates. Consequently, the ES model can function in situations where very few representative timing measurements are available, unlike other ACET analysis approaches which require a large number of measurements, and a lengthy off-line analysis period to derive their final estimates.

However, the sacrifices made to ensure the ES model can perform within a limited operating environment incur costs in terms of its overall predictive performance. The use of a smaller set of measurements to derive ACET estimates may mean that seasonal or cyclical trends with the data are ignored, or repeating periods of increased volatility in the measurements are missed. Similarly, one-off trends within the data cannot be processed effectively within the ES model, for example, the timeliness of some software processes can increase linearly and then level off as arrays are filled, and other data structures are created in memory. An ideal ACET analysis method would automatically determine the stable operating pattern of a particular process, and use that as the basis for more representative average-case estimates. Unfortunately, the ES model cannot make this assessment using the limited set of measurements it has at its disposal. The restricted sample size also prevents the ES model from assessing the significance of any deviations within timing behaviour, i.e., whether a large increase or decrease in timeliness is statistically significant or indicative of a change within the underlying system. Lastly, while the ES model provides an average-case estimate of the next immediate timing measurement, a more longer term forecast would enable more pre-emptive adaptations to correct perceived future performance issues.

3.4 Worst-Case Analysis

Worst-case timing analysis seeks to determine the likely maximum execution time performance of software. In common with the average-case timing analysis processes described earlier, TimePredict uses a measurement-based approach to derive a worst-case execution time (WCET) bound

for the current configuration of the software. However, unlike the average-case timing analysis, the WCET value for the software may not become immediately apparent using a measurement-based approach [Colmenares et al., 2008]. This extreme worst-case timing behaviour will rarely (if ever) be exposed through repeated timing measurements, therefore a pessimistic statistical model must be applied to the existing timing data in order to encapsulate potentially unobserved timing behaviour. Statistical modelling techniques can be applied where the complexity of the underlying system precludes a formal analysis, or where estimates must be made based on incomplete data. With dynamically-adaptable software systems, the current configuration of the software cannot be anticipated from a static context, so any worst-case timing analysis must be performed at run-time, and refreshed after each adaptation to the system.

TimePredict uses statistical modelling techniques more commonly found in materials science [Castilloa et al., 2006], environmental engineering [Alvarado et al., 1998] and the analysis of financial markets [Rosenberg and Schuermann, 2006], to predict the worst-case performance of dynamically-adaptable systems. The worst-case statistical models, collectively known as extreme-value distributions, are a specialized type of statistical distribution that are commonly employed to estimate rarely occurring events [Coles, 2001]. Extreme-value distributions are heavy-tailed, since greater emphasis is placed on the tail behaviour of the distribution compared to the Normal (Gaussian) distribution. This allows extreme-value distributions to be applied to the prediction of rare events, such as worst-case execution times, based on the more typical timing behaviour of the system [Edgar, 2002]. However, a set of measurements are first required to fit these statistical distributions to the underlying process. Until a sufficient number of measurements are available that expose the likely range of software execution times, another non-distribution timing analysis process must be initially applied to estimate the WCET.

3.4.1 Initial Worst-Case Heuristic

When the software begins execution, or restarts execution in the wake of a functional adaptation to the system, the previously collected timing measurements may not represent the new configuration of the system. A fresh set of timing measurements is typically required to update the ACET and WCET estimates of the software, however these may only be generated at run-time as the software executes. A period may occur, immediately following an adaptation, where an insufficient number of timing measurements are available to perform a detailed worst-case

statistical analysis of the software timing behaviour.

Similar to the ES model used by TimePredict for ACET estimates, a worst-case heuristic provides an intuitive timing estimate where there is insufficient timing data to perform a more detailed analysis. The intuitive worst-case estimate produced uses the observed worst-case value from the set of previous timing measurements, to forecast a WCET bound. In cases where the timing behaviour of the software is well-defined, the heuristic provides a slightly pessimistic timing estimate that will closely match the expected worst-case behaviour. In all other cases, i.e., where the software timeliness is highly volatile, or follows no discernible pattern initially, the estimate produced by the heuristic will tend towards the maximum observed execution time, recorded thus far. A multiplier is used to increase the WCET bound, using the number of times previous estimates had been exceeded over the previous n measurements. This multiplier, presented in Equation 3.6, provides a readily accessible WCET estimate when there are insufficient measurements available to make a more detailed statistical analysis. The multiplier was designed to rapidly increase in cases where there are few measurements and many violations of the WCET bound, and then decrease over time, tending towards the observed WCET measurement if no further violations of the bound occur.

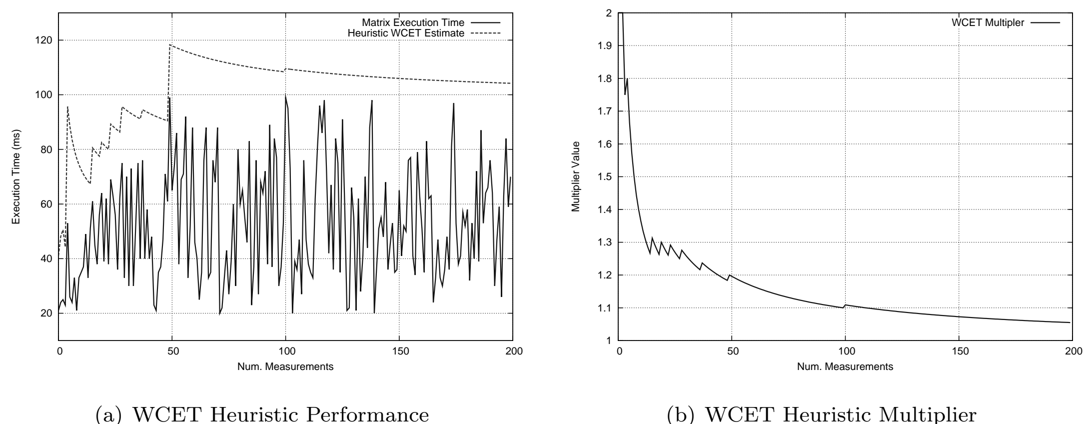
Unfortunately, this heuristic method lacks any associated level of confidence in the correctness of its estimate. For example, there is no know way of estimating the likelihood of the next timing measurement exceeding the heuristic WCET bound, aside from perhaps an evaluation of its historic accuracy. This limits its application to periods when there are too few measurements available to provide a better prediction, such as immediately after an adaptation, or when the software begins execution initially. However, the WCET heuristic may prove to be instructive within the context of feedback into the software adaptation process, until a better estimate can be produced. The predicted WCET bound ($\hat{W}_{n+1}$) takes the initial timing measurement (t_0) as its first estimate. When more measurements are made available, the heuristic is applied, i.e.,

$$\hat{W}_{n+1} = \begin{cases} t_0, & \text{When } n = 0, \\ ((n+m)/n)w_n & \text{Otherwise} \end{cases} \tag{3.6}$$

The observed worst-case timing value is described by w_n, and can change as more timing measurements are added, i.e., when larger timing measurements are reported. The observed worst-case timing value can be defined as the maximum value in the current set of timing mea-

surements. The cardinality of the set of timing measurements is defined as n. The parameter m defines the number of times the observed worst-case timing value (w_n) has been exceeded by a new timing measurement, i.e., $t_{n+1} > w_n$. The value of m will always be less than n, so that the value described by $(n+m)/n$ will be greater than 1. This value provides a multiplier to generate a more pessimistic timing bound using the current observed worst-case timing measurement.

Figure 3.4 shows the performance of the WCET heuristic, and the increase in the WCET multiplier. The WCET heuristic estimates were produced using sample time-series data taken from measuring the execution time of a matrix multiplication benchmark [Arndt et al., 2009], running on an embedded Sun SPOT device.



(a) WCET Heuristic Performance

(b) WCET Heuristic Multiplier

Fig. 3.4: WCET Heuristic Performance.

Figure 3.4(a) illustrates the WCET estimate growing quickly over the initial measurements, where the WCET bound is exceeded several times initially. Once this WCET bound is sufficiently larger than the measurements it begins to recede until is approximates the largest observed WCET measurement. Figure 3.4(b) shows the multiplier value used to generate the WCET.

The advantages of this heuristic are its ability to generate pseudo-WCET bounds using a limited number of timing measurements, i.e., timing estimates without any associated statistical confidence. The accuracy and precision of the WCET bounds produced will be dependent on the volatility and complexity of the underlying timing behaviour of the software, however the heuristic is presented as an initial WCET timing analysis method, to be used only until sufficient timing measurements permit the creation of statistically-backed WCET bounds. In practice, this worst-case heuristic is useful in cases where the software must be quickly configured, or tasks

scheduled, without an appropriate high-confidence estimate being available. This is especially relevant within dynamically-adaptable systems, where a new configuration of the software may have been deployed with the expectation of a particular worst-case execution time, only to find this initial expectation badly at odds with its observed behaviour. Rather that wait until a large number of timing measurements have been made, and a statistically-assured worst-case estimate produced, the heuristic estimate allows the adaptation manager to quickly re-adapt (or roll-back) any poorly performing adaptations to the system [Fritsch and Clarke, 2008].

3.4.2 Generalized Extreme Value Model

The generalized extreme value model (GEV) is a family of continuous probability distributions, noticeable for being 'heavy-tailed'. A heavy-tailed probability distribution contains slightly more volume in the extremities (tails) of the distribution, and is defined as having a slower than exponential rate of decay of its complementary cumulative distribution [Kotz and Nadarajah, 2000]. In essence, extreme value distributions are less clustered towards an average value when compared to the more frequently used Normal (Gaussian) distribution.

The TimePredict approach uses the GEV model to provide WCET bounds, when sufficient timing measurements are available to successfully fit a statistical distribution, and model the likely (unobserved) worst-case timing behaviour. The GEV model is comprised of the Fréchet, Gumbel and Weibull distributions, each distribution having a slightly different shape, allowing TimePredict to select the most appropriate distribution to represent (fit) the underlying timing measurement data.

3.4.2.1 Gumbel Distribution

The Gumble distribution, often referred to as a Type I extreme value distribution, differs from both the Type II (Fréchet) and Type III (Weibull) distributions in that it has no associated shape parameter, i.e., the curve described by the probability density function (pdf) does not change its essential shape, but may change its location or scale. The pdf is a function that describes the probability of a randomly selected value occurring at a particular point within the range of the distribution [Evans et al., 2000]. The probability of a random variable falling within a specified range of the distribution being given by the integral of its density between those points, i.e., the area under the curve illustrated in Figure 3.5. To simplify the description of the GEV model,

the equations describing the behaviour its various distributions omit their scale and location parameters, instead presenting a 'standard' distribution of each statistical model. The pdf for the Gumbel distribution can be defined as,

$$f(x) = e^{-x}e^{-e^{-x}} \tag{3.7}$$

Where $f(x)$ represents the pdf for a particular value (x) along the horizontal axis (see Figure 3.5). The cumulative distribution function (CDF) describes the probability that a random value taken from a Gumbel distribution will occur before a specific value (x). The CDF is the integral of the pdf, and effectively adds the probability densities over the entire range of the distribution. The CDF is used extensively within TimePredict to find the value at which 99% of the fitted timing measurements are likely to have occurred, i.e., the probability of a value exceeding this bound would be 1% if the data fits a Gumbel distribution. The CDF for the Gumbel distribution can be defined as,

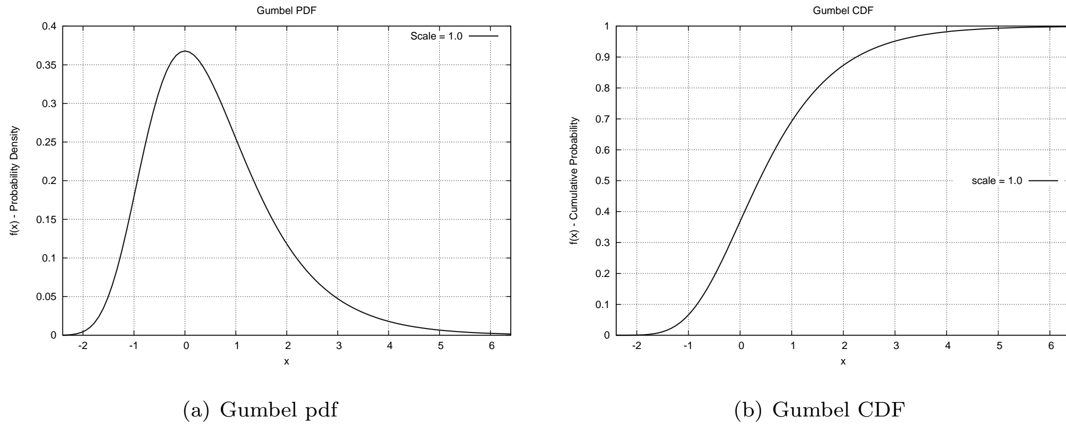
$$f(x) = e^{-e^{-x}} \tag{3.8}$$

The pdf and CDF for the Gumbel distribution are illustrated in Figure 3.5. The location parameter and scale parameter for the distribution is set to 0 and 1 respectively, to provide a 'standard' distribution that can then be fitted to the timing data to produce a WCET estimate. Even though the x-axis within Figure 3.5 includes negative values, these would be translated and scaled along the axis to positive values, once the model has been fitted to the data.

Statistical models must be fitted to the underlying data to provide a valid estimate. This process is described in more detail in Section 3.4.3, but in essence the underlying data is ranked from smallest to largest, and the distribution CDF is scaled and transformed until the closest match is found. If the correspondence between the data and the fitted distribution is deemed to be close enough, values may then be read from the CDF of the distribution to determine a value with a particular probability of exceedance.

3.4.2.2 Fréchet Distribution

The Fréchet distribution, or Type II extreme value distribution, differs from the Gumbel distribution in that it has an associated shape parameter that makes the distribution more or less biased towards increasing values of x. The pdf for the Fréchet distribution can be defined as,



(a) Gumbel pdf

(b) Gumbel CDF

Fig. 3.5: Gumbel probability distribution.

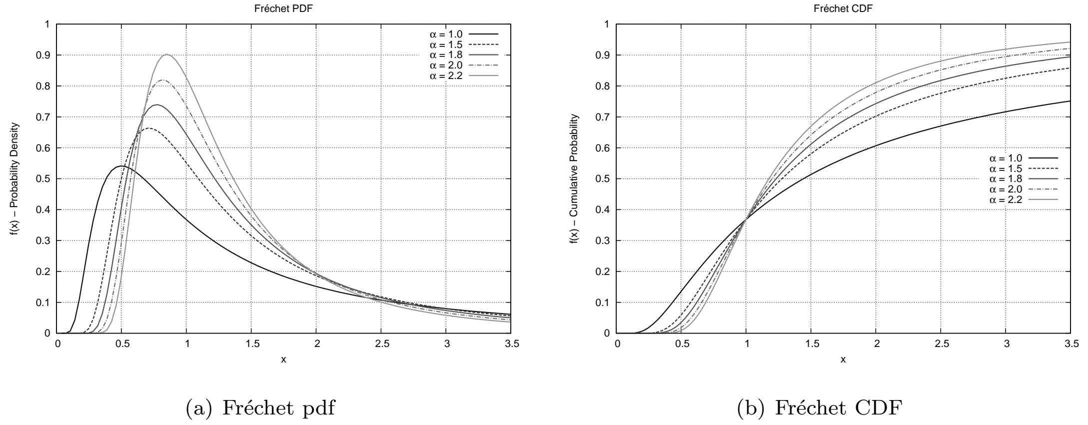
$$f(x) = (\alpha x^{-\alpha-1})/(e^{x^{-\alpha}}) \tag{3.9}$$

Where α provides the shape parameter for the distribution. The Fréchet distribution pdf and CDF are both defined for all values greater than zero, with a progressively increasing bias towards larger values of x as α increases. In addition, the curve of the pdf/CDF grows steeper within greater values of x, essentially restricting more of the volume of the curve to a smaller area. The CDF of the Fréchet distribution is given by the equation,

$$f(x) = e^{-x^{-\alpha}} \tag{3.10}$$

The effects of increasing the shape parameter can be clearly observed in Figure 3.6, showing how increasing values of α increase the height of the curve and its bias towards the right-hand side of the graph.

Fitting statistical distributions containing a shape parameter are more complex, since more than simple scaling and transformation needs to the take place to match the pdf/CDF to the underlying data. Typically, a range of values for the shape parameter are initially applied, and progressively refined until an appropriate value found. When the statistical model has been appropriately fitted, the value of x matching the required probability is taken from the CDF, e.g., finding the x value corresponding to 99% by solving the equation $0.99 = e^{-x^{-\alpha}}$, where the value for α is known.



(a) Fréchet pdf (b) Fréchet CDF

Fig. 3.6: Fréchet probability distribution.

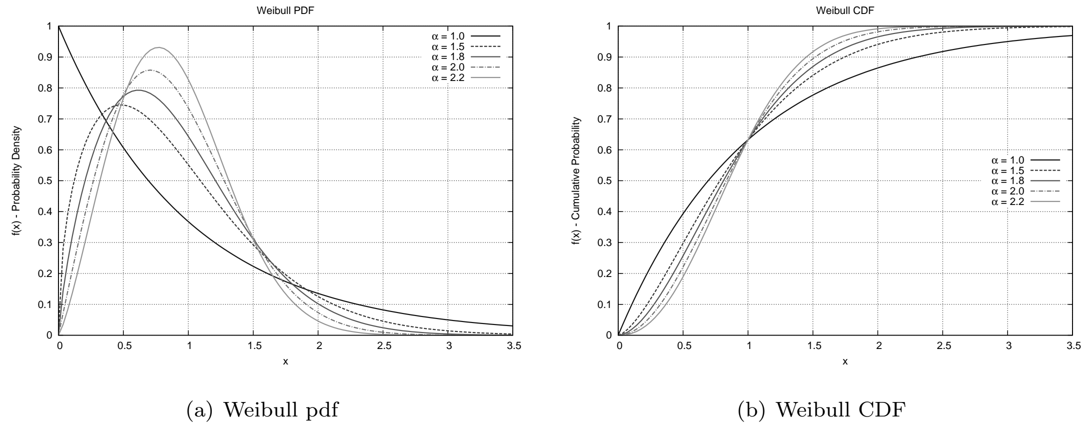
3.4.2.3 Weibull Distribution

The final distribution within the GEV model is the Weibull distribution, also sometimes termed the Type III extreme value distribution. Whereas the Fréchet distribution is undefined at zero, the standard Weibull distribution, i.e., with no scaling or location parameters, is defined from 0 to ∞ (infinity). In common with the Fréchet distribution, the Weibull distribution is defined using a shape parameter (α). The pdf for the Weibull distribution is defined as,

$$f(x) = \alpha x^{\alpha-1} e^{-(x^{\alpha})} \tag{3.11}$$

However, unlike either the Fréchet or the Gumbel distributions, the Weibull distribution pdf may take on one of two distinct shapes - an exponentially decreasing curve when $\alpha = 1$, and a skewed bell-shaped curve for all $\alpha > 1$. These two distinct distribution shapes are illustrated in Figure 3.7.

The exponentially decreasing curve ($\alpha = 1.0$) and the bell-shaped curve ($\alpha > 1.0$) can both provide a good statistical model for the tail behaviour within a process. However, selecting one particular shape will depend largely on the range and number of the underlying measurements. The Weibull distribution may be fitted to the data, starting with an initial estimate for the shape parameter, and progressively altering this value until the most appropriate fit is found. The Weibull CDF provides the means of both fitting the distribution to the data, and selecting the appropriate point on the curve for an estimate with the specified level of confidence. The



(a) Weibull pdf

(b) Weibull CDF

Fig. 3.7: Weibull probability distribution.

CDF may be defined as,

$$f(x) = 1 - e^{-(x^{\alpha})} \tag{3.12}$$

The point of convergence within the CDF, i.e., when $x = 1.0$, provides a useful anchor point when fitting the distribution. At this point, regardless of the shape parameter, the probability of exceedance is 63.2% (since $1 - e^{-(1.0)} = 0.632$). This allows the distribution to be scaled and translated along the horizontal axis, until the CDF of the distribution matches the frequency distribution of the underlying data. This initial fitting can then be refined through modification of the shape parameter, as well as more subtle increments to the scale and location.

3.4.2.4 Generalised Pareto Distribution

The Generalised Pareto Distribution (GPD) is a power law probability distribution and although not part of the GEV model, it may be used to provide a good statistical model for exceedances. An exceedance is, in essence, the sum of the number of measurements within a data set that are greater than a specified value. By incrementally increasing this specified value, a frequency distribution can be generated that shows the exceedances over a range of values. For example, within a group of 100 people, all may be taller than 1.5m, 50 people may be taller than 1.75m and only 10 people taller than 1.9m. Reducing the measurement interval to 0.1m ranges would probably reveal a curve similar to the Pareto pdf in Figure 3.8. For processes such as software

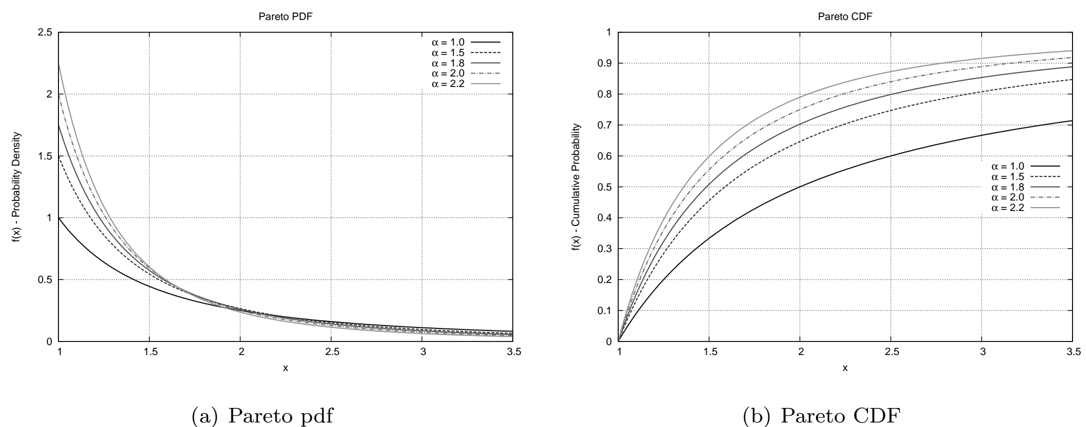
execution times, the probability of a randomly selected measurement exceeding a particular time limit tends to exhibit a similar exponentially decreasing series of values. The pdf for the Pareto distribution can be defined as,

$$f(x) = (\alpha)/(x^{\alpha+1}) \tag{3.13}$$

Similarly to the distributions presented within the GEV model, the Pareto distribution provided in (3.13) does not contain scale or location parameters. Instead, the pdf/CDF defines a standard distribution, with a scale of 1 and a location of 0 (i.e., no scaling and no translation along the horizontal axis). The CDF can be found using the equation,

$$f(x) = 1 - (1/x)^{\alpha} \tag{3.14}$$

The Pareto distribution describes a single shape, that of an exponentially decreasing curve. The best-case fit for a Pareto distribution is when the underlying data contains many small value measurements clustered around a limited range, and fewer large measurements that decrease in their frequency with size. The shape of the Pareto pdf and CDF is illustrated in Figure 3.8. Fitting the Pareto distribution is done in much the same way as any of the previous GEV distributions. A range of values for α are initially specified, and refined until the most appropriate fit is found. The estimate is then taken from the CDF, which can be described using the equation,



(a) Pareto pdf

(b) Pareto CDF

Fig. 3.8: Pareto probability distribution.

3.4.3 Model Fitting

Both the GPD and GEV statistical models may only generate predictions once they have been appropriately scaled and transformed to match the underlying data. This process, known as model fitting, is achieved through discrete adjustments of the parameters that define each distribution. The steps required to fit a statistical model can be summarized as,

1. Create a frequency distribution from the underlying data.
2. Add each value from the frequency distribution, to produce a cumulative distribution.
3. Select one of the GEV distributions, and adjust its shape, scale and location parameters until it closely matches the cumulative distribution of the data.
4. Generate the goodness-of-fit test statistic (see Section 3.4.3.1)
5. If the test statistic is less than the critical value, select this distribution as the predictive model.
6. Otherwise, iterate through the remaining GEV statistical distributions fitting and testing each in turn.
7. If no appropriate fitted distribution can be found, use the WCET heuristic estimate.
8. Assuming there is a suitable fitted distribution, select the required level of confidence in the estimate, i.e., the likely reliability of the estimate at encapsulating measurements within the fitted distribution. Typically this is set to some value greater than 99%.
9. Find the value on the CDF of the fitted distribution that corresponds to this desired level of confidence, e.g., the x value corresponding to $f(x) = 0.99$ for the CDF.
10. Return this x value as the worst-case estimate.

The first step requires the dataset of timing measurements to re-ordered as a frequency distribution. Since the measurement data is already ordered from smallest to largest within the WCET array, all that is required is to specify an interval, and count the number of measurements within each interval over the entire range of the dataset. The resulting frequency distribution (as per the example in Figure 3.1), is used as the template on which the statistical models are

then fitted. Each statistical distribution is fitted in turn, by iteratively adjusting the shape, scale and location parameters that define its CDF, so that it closely matches the frequency distribution of the data. Once each distribution has been fitted, a goodness-of-fit test determines which distribution provides the closest match for the data. This is then used to generate an appropriate timing estimate (see Section 3.4.3.2).

3.4.3.1 Goodness of Fit

Whereas fitting a particular distribution to the data is simply a matter of changing the parameters that govern its pdf/CDF, a separate process is required to evaluate how closely the fitted distribution matches the underlying data. A goodness-of-fit test evaluates whether a particular distribution fits the underlying data sufficiently well to be considered as a good representative model. The notion of 'sufficiently well' within the context of statistical testing, implies that there is no strong evidence to suggest that the data comes from anything but the fitted distribution [Kotz and Nadarajah, 2000]. This initial assumption, known as the null hypothesis, must be rejected in order to reject the fitted distribution. The test itself is performed by comparing a calculated goodness-of-fit metric (the test statistic) with an established statistical limit. A suitable level of confidence in the goodness-of-fit test can be selected, to determine the appropriate statistical limit, and accept only fitted distributions of a sufficiently close match to the data.

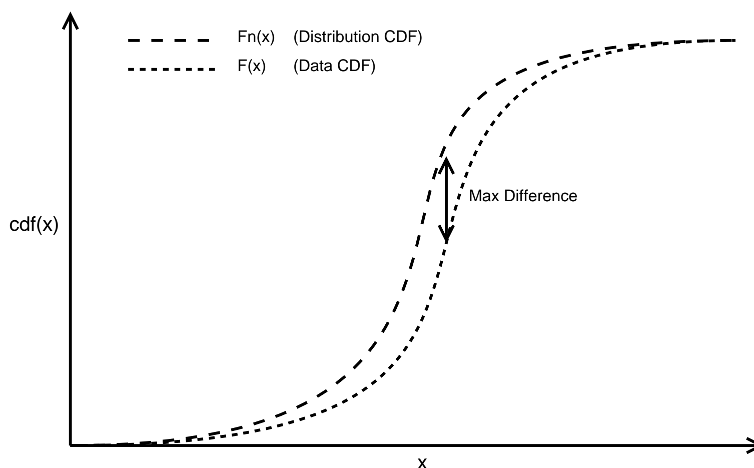


Fig. 3.9: Analysis of the difference between Data and Distribution CDFs.

TimePredict uses the Kolmogorov-Smirnov goodness-of-fit test [Massey, 1951], to determine

which of the fitted distributions best represents the dataset comprised of extreme timing measurements. The Kolmogorov-Smirnov test, also known as the KS test, is a minimum distance test (illustrated in Figure 3.9) that determines the goodness of fit for a particular distribution based on the sum of the absolute distance of its CDF from the frequency distribution of the dataset [Young, 1977]. TimePredict implements the KS test to select the fitted distribution closest to the frequency distribution of the observed execution time behaviour, since this could be deemed to provide the best mathematical model to predict the WCET bounds for the software.

The critical value for the KS test is calculated by finding the supernum of the absolute difference between the fitted distribution ($F_n(x)$) and the CDF of the data ($F(x)$). The supernum can be described as the maximum vertical displacement between the data and the fitted distribution. Figure 3.9, illustrates an example of this maximum vertical displacement between two example CDFs. This test statistic (D_n) is found by iterating though both CDFs, and reporting the maximum difference found. This test statistic can be calculated using the equation,

$$D_n = \sup_x |F_n(x) - F(x)| \tag{3.15}$$

where $\sup_x$ is the supernum of the difference. Once D_n has been determined, it must be compared to a specific critical value for a two-sample KS test (i.e., where two CDFs are compared). These critical values are presented, for differing degrees of stated confidence, in Table 3.2.

α_c	D_{crit}
0.1	1.22
0.05	1.36
0.01	1.63
0.005	1.73
0.001	1.95

Table 3.2: Confidence intervals and critical values for the Kolmogorov-Smirnov two-sample test.

If the test statistic is less than the critical value in the table, the fitted distribution cannot be found to be sufficiently different from the underlying data, and hence the fitted distribution is accepted. The values in Table 3.2 are taken from a similar table in [Young, 1977], as well as [Lindley and Scott, 1995], and offer critical values for the KS test, defined by confidence level

and sample size. The degree of confidence is provided by $1 - \alpha_c$ in the table, i.e., where an α_c value of 0.01 corresponds to a level of confidence of 99% in the KS test.

3.4.3.2 Generating Estimates

When the KS test has indicated the most appropriate statistical distribution for the existing timing data, the chosen distribution can then be used to generate a WCET bound for the software. The CDF for the selected distribution is used to provide a WCET bound, based on a pre-defined level of confidence. This level of confidence is a measure of the probability of a particular timing measurement occurring, assuming that the distribution closely corresponds to the set of previously observed timing measurements, and the underlying process, i.e., the software behaviour, does not change. To generate a WCET bound, the CDF is solved for x, e.g., as in the case of the Weibull distribution CDF,

$$f(x) = 1 - e^{-(x^{\alpha})}$$

where both $f(x)$ and α are known, and solving for x provides the WCET bound. The probability of a timing measurement being encapsulated by this WCET bound is provided by $f(x)$. By default TimePredict sets this probability to 0.99, i.e., a 99% probability that the next timing measurement will be less than the WCET bound. In practice, the estimate probability achieved by TimePredict will be much higher than 99%, since it actively replaces smaller timing measurements with larger ones within the array of WCET data, resulting in a slightly more pessimistic WCET bound.

3.5 Validation and Feedback

TimePredict can produce a continuously updated ACET/WCET timing estimate, as more timing measurements become available. The accuracy and precision of these estimates is recorded, and used as a means of selecting the most appropriate average-case and worst-case predictive model based on recent performance. Two low resolution predictive methods, namely the ES model and the WCET heuristic, are more appropriate when there are few timing measurements available, such as immediately following an adaptation to the system.

When sufficient timing measurements have been recorded, the more detailed statistical models

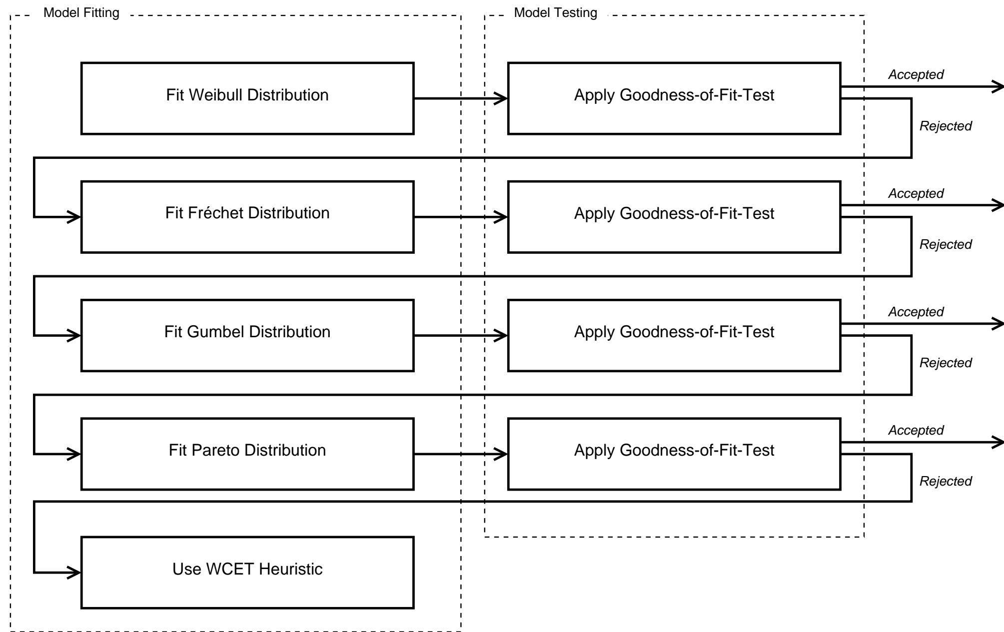


Fig. 3.10: WCET model fitting process.

(GEV/GPD respectively) can be used to create timing estimates with a greater level of confidence, by evaluating a larger cross-section of the timing measurements. However, since these statistical models are grounded in a number of assumptions, e.g., a constant variance/mean within the measurements, the models may become imprecise or inaccurate if these assumptions are violated. The usage of both model-based and more heuristic-based estimates within TimePredict ensures that an estimate is provided, in all circumstances where at least one valid timing measurement has been recorded. Figure 3.10 illustrates the WCET model fitting process, showing how the GEV/GPD models are each fitted and tested in turn, and if none provide a sufficiently close fit to the data, the estimate is produced using the WCET heuristic. In all circumstances where there is at least one representative timing measurement, TimePredict can generate both ACET and WCET estimates.

TimePredict can be configured to select the most appropriate WCET model using one of two strategies, a periodic comparison and ranking of the goodness of fit across all the GEV/GPD distributions, or a frequency ordered ranking according to a minimum goodness of fit threshold. In the first case, TimePredict performs a goodness of fit for each distribution at set intervals, and ranks the distributions in order of those that best fit the available data. This strategy has the

advantage of ranking the GEV/GPD distributions in terms of their likely predictive performance, however, it requires additional processing at every update interval to refresh this ordering. The second strategy adopts a type of limited analysis approach, in that a critical value is specified in advance for the KS test, and the first distribution to achieve this fit is selected. The frequency of these selections over time are then used to rank the various distributions. The advantage of this approach is that it reduces the processing overhead on the system, however, because it does not exhaustively test each distribution, a slightly better fit for the data may not be found unless the current top-ranked distribution fails to meet the critical value for the KS test.

The percentile used by TimePredict to determine the WCET estimate from a fitted model, is set by default to be the 99th. While this is arbitrarily selected as being indicative of a bound unlikely to be surpassed during normal operation, maybe being exceeded on average 1% of the time if the distribution matches the data. Although this 1% probability of exceedance has the potential to occur, the dataset underlying the WCET models is bias towards large measurements, i.e., the limited storage space for measurements means that smaller values are progressively replaced by larger ones. This in turn impacts on the probability of exceedance, in that TimePredict effectively creates a worst-case subset from the overall measurement data, in effect decreasing the probability of exceedance for the resulting WCET estimate. Although the required maximum allowable probability of any measurement exceeding this estimate may be set as high as 10^{-12} within some operating environments [Bernat et al., 2003], for the purposes of dynamically-adaptable software with changeable operating characteristics, the 99th percentile should be sufficient to be termed as a high-confidence estimate. Any non-pessimistic estimate with a more extreme probability of exceedance would require more measurements (which may not be available at all times), greater overhead in terms of data storage and processing, and possible implementation restrictions to limit the effects of adaptation on the underlying system. As such, TimePredict favours the default 99th percentile as the basis for worst-case estimates, in that it encourages functional expressiveness and adaptability, over limiting the software to ensure greater predictability.

However, the timing estimates generated by TimePredict in themselves will do little to optimize the performance of the software, unless they are used as an input into the adaptation selection process. The design of the TimePredict approach envisages timing estimates being used within an Adaptation Manager function (as in Figure 3.2), that evaluates, selects and

implements functional adaptations at run-time, based on the execution time behaviour of the current configuration of the system. Comparing the current/recent performance of the system against a set of QoS requirements, would enable the system to react to subtle changes in its operating environment, and adjust its functional scope for the prevailing conditions. A time-optimizing dynamically-adaptable system would be capable of changing its functional makeup, to maximise the system response time across a large number of potential operating scenarios. The next chapter presents a number of these scenarios, and shows how informed run-time functional modifications, based on estimates provided by TimePredict, can dynamically optimize the performance of the system.

Chapter 4

TimePredict Implementation

Nos numerus sumus et fruges consumere nati.
We are but numbers, born to consume resources.
Horace (Quintus Horatius Flaccus)

The design of the TimePredict approach, presented in the previous chapter, must be implemented in software and tested within a live operating environment before any assessment can be made of its predictive performance. In addition to the realization of the various statistical models that comprise the TimePredict approach, the timing analysis must be shown to operate effectively within a highly-variable environment, on a resource-constrained embedded device. Since this run-time timing analysis process must both generate highly accurate timing estimates, as well as minimize its impact on the normal execution of the system, its correct implementation is critical to the overall goals of this thesis - namely, to provide a precise, accurate run-time timing analysis process suitable for dynamically-adaptable software running on resource-constrained and embedded devices.

The principal aims of this chapter are to describe the target operating environment, present the implementation of the timing measurement process, and outline the various statistical methods that form the core of the TimePredict approach. In order to initially adjudge the effectiveness of these predictive models, a test framework is introduced that enables the simulation of a number of potential timing behaviours, as well as imitating the effects of sudden functional adaptations on software timeliness. When TimePredict was verified within this virtual operating environ-

ment, it was then deployed within a live implementation of a dynamically-adaptable system, running on a resource-constrained embedded device. The implementation of the various features of TimePredict are presented in this chapter, including the role of run-time timing analysis in the initiation and selection of functional adaptations to the system. Lastly, the underlying theoretical assumptions and limitations of the various predictive models are validated against a realistic application scenario executing within a live environment.

4.1 Operational Implementation

TimePredict is divided into two appreciably distinct logical parts, one concerned with forecasting the average-case software timeliness, and the other with deriving a worst-case performance estimate. This logical division is necessary since the average-case and worst-case analysis methods evaluate fundamentally different properties of the underlying software timing behaviour. Consequently, both analyses have their own unique set of requirements on the storage of timing measurement data. For example, the average-case ES model requires a small set of recent timing measurements ranked in order of their occurrence, whereas the GEV worst-case model depends upon a slightly larger set of measurements, ranked in order of their size and frequency. To facilitate both the average-case and worst-case analyses, TimePredict maintains two separate data structures, updating each at run-time with the latest software timing measurements. The predictive models, described in the previous chapter, are then applied to their respective data-sets to generate either an ACET or WCET timing bound.

Figure 4.1 illustrates the various classes that comprise the TimePredict approach. The average-case and worst-case analysis components are separated, with each predictive model reporting their timing estimates to a main `TimePredict` class. The `TargetSoftware` class represents the class or function used as the basis for the timing measurements within `TimePredict`. Timing measurements are taken using the `TimingListener` interface, which instruments the code with a `timerStart()` and a `timerStop()` function to record the interval between two consecutive time-stamps. The time-stamping functionality itself is encapsulated within the `TimingEvent` class, and provides further functionality to calculate the interval in milliseconds between two specified time-stamps.

The `TimingUpdate` class refreshes the timing information stored in the respective arrays within the `ACETData` and `WCETData` classes. These data structures are used to provide the worst-

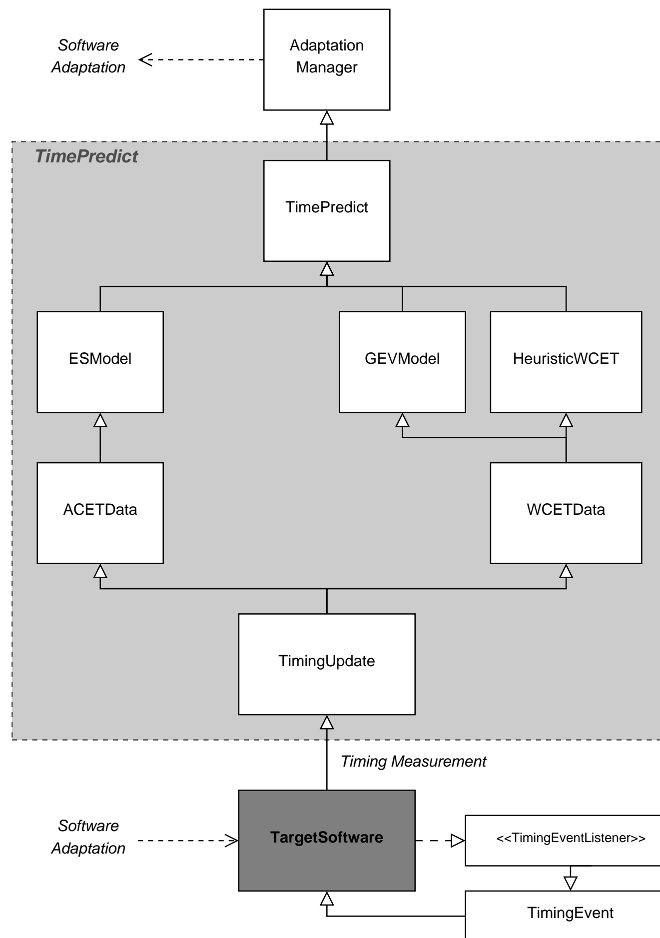


Fig. 4.1: TimePredict class diagram.

case and average-case predictive models with timing measurements, and can be dynamically re-sized during execution. The implementation of these arrays, and the algorithms used to update their stored timing measurement data are described in more detail in Section 4.2.1.1. In addition to updating these classes with new timing measurement data, `TimingUpdate` also performs a simple evaluation of the accuracy and precision of the previously generated timing estimates for each predictive model, by comparing the old ACET and WCET bounds with the latest timing measurement data. Where a particular predictive model is found to have become increasingly inaccurate, the `TimingUpdate` class informs the `TimePredict` class to select an alternate predictive method.

The `ESModel` class implements the Exponential-Smoothing model for average-case timing estimation. Similarly, the `GEVModel` and `HeuristicWCET` classes implement the model-based and heuristic-based worst-case timing analysis. The estimates from both average and worst-case models are forwarded to the `TimePredict` class, which selects the most appropriate estimate (based upon the evaluation performed by `TimingUpdate`) and makes a revised timing forecast available to the Adaptation Manager. Based on changes in the timing estimates produced by `TimePredict`, this Adaptation Manager can then decide to modify the `TargetSoftware` in order to optimize its timing behaviour. If an adaptation is scheduled, the `TimePredict` class is informed, so that previously recorded timing data can be removed, and updated timing estimates produced for the newly-adapted system.

While the `ESModel` and `HeuristicWCET` classes iterate over a small number of timing measurements, the `GEVModel` implementation relies of a relatively larger set of representative measurement date to fit and validate the model parameters. The TimePredict approach is unique in that is assumes there will be an insufficient, or otherwise very limited set of timing measurements available to the GEV model initially. Consequently, the WCET heuristic model is provided to offer a basic worst-case estimate until enough measurements have been recorded to fit a GEV model, and begin generating estimates. Since the memory resources of the target operating systems will be limited, it may not be feasible to record every timing measurement. Since only relatively larger timing measurements are of direct interest to the WCET analysis, TimePredict records the number of occurrences of timing values between a specified range of values. Where the clock resolution is 1 millisecond, TimePredict records the number of measurements in steps of 1 ms between the largest previously observed measurement, and some lesser value. The range of measurements included in this timing frequency distribution is referred to as the lag length, and can be adjusted by the user to facilitate greater accuracy when a larger lag length is selected, or less memory overhead when a smaller lag length is used. By measuring the occurrence of timing measurements over a greater range of values, a more accurate model may be fitted to the data, however, by focusing on the worst-case 'tail' measurements only, a WCET bound may still be produced, albeit with a possibly greater than necessary distance between the observed WCET and the estimate (i.e., a pessimistic estimate). The lag length is therefore used to focus the priorities of the `GEVModel` class, greater precision in generating estimates, or a reduced impact on the underlying system in terms of memory and processing resources used.

4.1.1 Target Operating Environment

The target operating environment is envisaged as being predominantly defined by the limited processing and memory resources of the underlying hardware platform, as well as the unanticipated nature of software adaptations to the system. The primary challenge estimating the timeliness of dynamically-adaptable systems, is the potentially large number of possible software configurations, each with their own performance characteristics. In addition, each software configuration can be enabled seemingly at random, based on unanticipated changes in the operating environment.

Measurement-based timing analysis produces its own set of challenges, especially when the supporting hardware is limited in terms of its processing power or memory, or during periods immediately following an adaptation when few timing measurements are available to analyse. A measurement-based timing analysis process must provide accurate estimates, even within highly-variable operating environments, while minimizing any impact on the 'normal' operation of the system. In addition, any unanticipated changes to the timing behaviour of the software must be met by corresponding changes to its timing estimates, without restricting or delaying any necessary functional adaptations. The impact the TimePredict approach may have on the target operating environment, and the resulting optimizations in its implementation that preempt any negative functional or non-functional interference, are discussed in more detail in Section 4.1.7.

4.1.2 Hardware Considerations

For the purposes of this thesis, it is assumed that TimePredict executes on a stand-alone resource-constrained device, provided with its own processor(s), memory and persistent storage. The definition of what constitutes a resource-constrained device may be subjective, however PDAs, mobile phones and wireless sensor motes have traditionally been considered as examples of these types of systems [Gal et al., 2006]. Laptops, tablet PCs and high-end smart-phones, while being highly mobile and wirelessly-connected devices, are more comparable to older desktop PCs, both in terms of processing power and operational limitations, than to any typical wireless sensor mote [Li et al., 2010]. The key characteristics of the target hardware environment are envisaged as incorporating a MHz-scale processor architecture, containing solid-state memory/storage not in excess of ten megabytes. The device itself should be powered from an internal rechargeable battery, and provide a wireless interface to support both data and operations & maintenance

(O&M) communication with an external server.

Fig. 4.2: Java Sun SPOT mote.

The hardware platform that provided the closest match to this set of high-level requirements, and the one ultimately selected as the basis for this work, was the Java Sun SPOT mote [Smith, 2007], as shown in Figure 4.2. Java Sun SPOTs (Sun Small Programmable Object Technology) are wireless-enabled motes, containing a single ARM processor, three-axis accelerometer, on-board light and temperature sensors, and running the Squawk Java VM (JVM) directly on the processor without a supporting OS [Smith et al., 2005]. The form factor for the mote illustrated in Figure 4.2, is approximately 41 x 23 x 70 mm, with each device weighing 54 grams.

The processor architecture used in the Sun SPOT mote, a 180MHz 32-bit ARM920T CPU, is a low-powered processor commonly found within mobile and embedded computing devices [Silven and Jyrkkä, 2007]. Each Sun SPOT is equipped with 512Kb RAM, and a 4Mb flash memory, providing a restricted, yet usable platform to run basic Java applications on top of the Squawk JVM. The lack of an operating system between this JVM and the underlying hardware does not negatively impact the performance of any Java applications executing on the device, and has instead been shown to out-perform an interpreted KVM running under Linux on a similar ARM

processor-based system [Simon et al., 2006]. In addition to the stand-alone motes, the Java Sun SPOT platform includes a base-station, to allow standard PCs to communicate wirelessly with any Sun SPOT mote within range.

A standard USB A-to-mini B cable can be used to update the Squawk JVM as well as upload software to the mote. In addition, this wired USB link provides a convenient means of monitoring the status of the Sun SPOT, by providing console output to a connected PC. Since dynamic software adaptation is achieved though the wireless deployment of Java class files to the mote (described in Section 4.2), monitoring the mote through a wired link avoids interleaving monitoring information with data traffic, that could potentially disrupt 'normal' execution on the device. In addition to the console output via the USB connection, the Sun SPOT mote provides eight user-programmable LEDs, a further eight electrically-driven I/O pins, and a number of on-board sensors. The various sensors available within the system can provide a measure of the acceleration, tilt, light and temperature within the device and are used to form the basis for the application scenario, described later in Section 4.3.

4.1.2.1 Wireless Communication

Java Sun SPOT motes and base-stations are identified by a unique 8-digit hexadecimal number, that serves as the permanent address of the device during communication. The IEEE 802.15.4 standard (ZigBee), provides the basic communications framework to enable motes, as well as any server-attached wireless base-stations, to initiate either stream-based and datagram-based communications within the 2.4GHz ISM radio band [Caicedo, 2006]. The wireless communications functionality is accessible through a number of Java APIs, and allows each mote a theoretic maximum bandwidth of 250 kilobits (kb). However, in practice the available bandwidth is often reduced due to bandwidth-sharing with other nearby motes, as well as signal attenuation caused by environmental factors. Again, under ideal conditions the wireless range of a Sun SPOT mote may be as much as 50m or 100m, but typically, the range is often limited to approximately 10m. For longer-distance communication, multi-hop communication is possible using the mote as a communications relay, where messages can be forwarded via other Sun SPOT devices to a target mote outside the immediate transmission range of the original sender. This multi-hop routing is supported by default within the Sun SPOT software stack.

```
//Setup a datagram connection for broadcast on port 110
DatagramConnection dgConnection = (DatagramConnection) Connector.
        open("radiogram://broadcast:110");

//Initialize a datagram
Datagram dg = dgConnection.newDatagram(dgConnection.getMaximumLength());

//Keep looping until discovered
while (!discovered)
{
        // Send the message as UTF-encoded string
        dg.reset();
        dg.writeUTF("DISCOVERY-REQUEST");
        dgConnection.send(dg);
        Utils.sleep(500);
}
```

Fig. 4.3: Sending a Datagram-based discovery request.

Typically, a discovery protocol must be implemented to identify the various Sun SPOT motes and base-stations within range. Luckily, the Squawk APIs support a datagram broadcast mode, that can send a repeating discovery request on a particular port defined within the range 0 to 255. This allows different communications channels to be associated with its own port number, and quickly identified by both sender and receiver. In the example code described in Figure 4.3, the discovery request is broadcast over port 110, with the request itself being a UTF-encoded string. The message is re-broadcast every 500ms, until a response is received (see Figure 4.4) and the Boolean flag `discovered` is set, thereby terminating the while loop.

Figure 4.4 outlines the code used to receive a datagram over a specific port number, such as a discovery request broadcast over port 110. Although the Squawk JVM supports both blocking or non-blocking datagram communications, the example above blocks further communication until a datagram is received over the specified port. Within a single threaded application, this blocking communication may be problematic, so to facilitate discovery, it is often more convenient to separate the transmitting and receiving functions into two separate threads, and have each run concurrently on the same mote, using Boolean flags for co-ordination. Once a discovery

```
//Setup a datagram connection for broadcast on port 110
DatagramConnection dgConnection = (DatagramConnection) Connector.
        open("radiogram://:110");

//Initialize a datagram
Datagram dg = dgConnection.newDatagram(dgConnection.getMaximumLength());

//Reset the datagram and wait to receive a message
dg.reset();
dgConnection.receive(dg);

//Get the address of the sender
String address = dg.getAddress();

//Print out the message and
String message = dg.readUTF();
System.out.println("Discovery Request: " + message + " from " + address);

//Send acknowledge message
dg.reset();
dg.writeUTF("DISCOVERY-ACK");
dgConnection.send(dg);

//Set the discovered flag
discovered = true;
```

Fig. 4.4: Receiving a Datagram, using a blocking receive function.

request has been received, an acknowledgement (`DISCOVERY-ACK`) needs to be sent, to confirm the reception of the original message.

While datagram-based communication is useful in discovering nearby Sun SPOT motes and base-stations, there is no guarantee packets will be correctly delivered, or received in the correct order. However, a `RadioStreamConnection`, illustrated in Figure 4.5, is provided within the Squawk API, that offers a more reliable alternative, allowing buffered stream-based communications between two named devices. The communication endpoints, i.e., the unique addresses of each device, must be known before a `RadioStreamConnection` can be initialized. Once the

```
RadiostreamConnection conn = (RadiostreamConnection)
        Connector.open("radiostream://" + baseStationAddr + ":112");
DataInputStream dis = conn.openDataInputStream();
DataOutputStream dos = conn.openDataOutputStream();
dos.writeUTF("MESSAGE");
dos.flush();
String response = dis.readUTF();
```

Fig. 4.5: Initializing and using a stream-based connection.

connection is opened, data input and output streams can be added to further assist data communication.

4.1.2.2 Sun SPOT sensors

The Sun SPOT motes provide a number of on-board sensors, that offer real-time information concerning the temperature, acceleration, tilt and ambient light observable by each mote. While largely unnecessary for the operation (or the concern) of TimePredict, the available sensor packages provide a convenient basis to motivate dynamic software adaptation within the Sun SPOT mote. Standard Java APIs allow access to the various sensors, and provide a up-to-date picture of the operating environment surrounding the mote. An application scenario, described in more detail in Section 4.3, outlines how a sensor application residing on the Sun SPOT, and reporting the various light, inclination, acceleration or temperature conditions, can provide an impetus for both dynamic software adaptation, as well as run-time timing analysis.

The function calls used to initialize and read the various on-board sensors are presented in Figure 4.6. The temperature and light readings are returned as double values, representing the current temperature in Celsius, and the light level in a range from 0 to 750, where 0 is complete darkness. The light level returns an average value taken over the previous 17ms, to account for non-constant light sources such as fluorescent lights. The tilt sensor gives the angle of inclination in radians, in all three dimensions. In Figure 4.6, only the inclination in the X-axis is read, however the other axes are accessible through the `getTiltY()` and `getTiltZ()` function calls. Lastly, the acceleration value provided by the on-board accelerometer gives the vector sum of the acceleration in all three axes. At rest, the base-line acceleration should equal 1.0 (gravity

```
//Initialize the sensors
IAccelerometer3D accSensor = EDemoBoard.getInstance().getAccelerometer();
ILightSensor lightSensor = EDemoBoard.getInstance().getLightSensor();
ITemperatureInput tempSensor = EDemoBoard.getInstance().getADCTemperature();

//Get the various sensor readings
double temp = tempSensor.getCelsius();
double light = lightSensor.getAverageValue();
double tilt = accSensor.getTiltX();
double accel = accSensor.getAccel();
```

Fig. 4.6: Taking sensor readings using the Squawk Java API.

only), however as the mote is moved or rotated, this acceleration value will register a change. Since the tilt value is read from the same accelerometer, changes in the inclination of the mote may result in corresponding changes in the acceleration reading from one or more axes.

4.1.3 Timing Considerations

Estimates of the timing behaviour of the software may be used as a means of selecting the most appropriate configuration of the system. Although outside the scope of this thesis, the timeliness of the software may provide an actionable QoS metric, as well as a threshold level that can prompt functional adaptations to the code. For example, in the case of a wireless sensor application, the maximum frequency of sensor updates will be largely determined by the execution time of the sensor software. Where there is some dependency between the sensor information provided by the application, and a remote actuator or server, the timeliness of updates may be crucial to ensuring correct operations over an extended period. In this case, a poorly performing sensor application may be determined by its failure to maintain a set update threshold, and can prompt functional changes to the underlying system to ensure this threshold is met. Likewise, the execution time of the application can lead to prioritization within the sensor functionality itself, such as changing the number or sensors reported, or the amount of local processing performed.

4.1.4 Resource Usage Considerations

The choice of a resource-constrained embedded device such as the Java Sun SPOT is desirable, since it provides an operating environment where both the availability of computing resources, and the execution-time performance of the software are critical for the correct operation of the system. In addition, a resource-limited operating environment offers a unique set of challenges in minimizing the memory footprint and processing overhead caused by any run-time timing analysis method. The execution-time measurement process, the size of the set of measurements stored on the device as well as the algorithms applied to generate the timing forecasts, must each be configured to forestall any unnecessary disruption to normal operations.

TimePredict avoids storing unnecessary or superfluous timing measurement data, by restricting the number of elements maintained within the data structures provided in the `ACETData` and `WCETData` classes (outlined in Figure 4.1). The maximum number of timing measurements stored at any one time is determined by the total memory available on the system and the required level of predictive accuracy for the statistical models. Since TimePredict is a measurement-based timing analysis approach, as more measurements are evaluated, the level of confidence in the resulting estimate tends to increase, and the precision of the ACET or WCET bounds improve [Hansen et al., 2009]. However, a greater number of timing measurements imposes additional costs on the system, in terms of the memory to store the measurements, and the added computational workload incurred by iterating over a larger dataset. By default, the arrays used within the `ACETData` and `WCETData` classes, restrict the maximum array size to 50 and 100 elements respectively. These values are selected to provide a reasonable balance between a required level of statistical confidence in the eventual timing estimate (up to 10^{-3}) against the need to have a small memory footprint and modest processing demands. The size of the various arrays used to store timing data can be configured by the user, and set to a greater or lesser value depending on the need for greater predictive accuracy or lower computation overhead (described in more detail in Section 4.1.4.2).

4.1.4.1 Memory Overhead

The `ACETData` measurement data is maintained within an ordered FIFO array, and is setup initially to store up to a maximum of 50 timing measurements. Once the array is filled, newer timing measurements progressively replace older data, so that only the 50 most recent measurements

are stored in the order of their occurrence. Even though the clock resolution of the standard Java platform restricts timing measurements to a millisecond-level accuracy, sub-millisecond accuracy is possible using dedicated timing hardware on the Sun SPOT device (see Section 4.1.5). As a result the ACET array is composed of a series of double values to record fractions of a millisecond. The memory overhead on the ACET array is 12 bytes for the array header and 8 bytes per element [Arnold et al., 2005], totalling 412 bytes in all.

In contrast, the `WCETData` class records a frequency distribution and cumulative distribution of the timing measurements, rather than the measurements themselves. A frequency distribution counts the number of occurrences of particular measurements, within a set interval, over an expected range of values. Since only the worst-case timing behaviour is required, the statistical models in the `GEVModel` class focus on the tail behaviour of this frequency distribution. A cumulative distribution, used to fit the models to the data, is also produced, by adding the sum of the terms of the frequency distribution into another array of the same size. Unlike the array of floating point numbers (doubles) stored in the `ACETData` class, the `WCETData` maintains two integer arrays, both limited to a maximum size of 100 elements. An integer rather than a double array is used, since the frequency distribution and cumulative distribution arrays use natural numbers to count the frequency spread of the measurement data. The memory overhead for an integer element is 4 bytes, with a 12 byte array header, meaning the two arrays each can consume an upper limit of 824 bytes within memory.

Initially both the average-case and worst-case arrays are undefined, but are initialized once the first timing measurement is generated by the system. To economize on the number of elements used to store timing data, TimePredict dynamically re-sizes each array when required, up to its pre-defined maximum size. This reduces the number of redundant elements maintained within each array, and ensures that only appropriate timing information is stored and analysed. Since the last element within the worst-case array is used to count the number of occurrences of the observed worst-case execution time, any new timing measurement that exceeds the observed worst-case value requires the smaller array elements to be removed/replaced to accommodate the new value.

The Sun SPOT SDK requires that all Java applications deployed to the device are first compiled into byte-code, and the resulting Java class files assembled into deployment files known as suites. These suites, using the compression of a JAR (Java ARchive) file, reduce the size of

the class files deployed to the mote by approximately 35% [Smith et al., 2005]. In addition to the reduced memory footprint of the deployed application, the Squawk JVM libraries executing on the mote consume only 80Kb of RAM, and 270Kb of flash memory, from a capacity of 512Kb and 4Mb respectively [Smith et al., 2005].

4.1.4.2 Processing Overhead

One of the principal tasks performed within TimePredict is the manipulation of arrays, albeit arrays of only 50 or 100 measurements. However, this initial setting is at the discretion of the system administrator, and can be increased where more accurate or precise timing estimates are required.

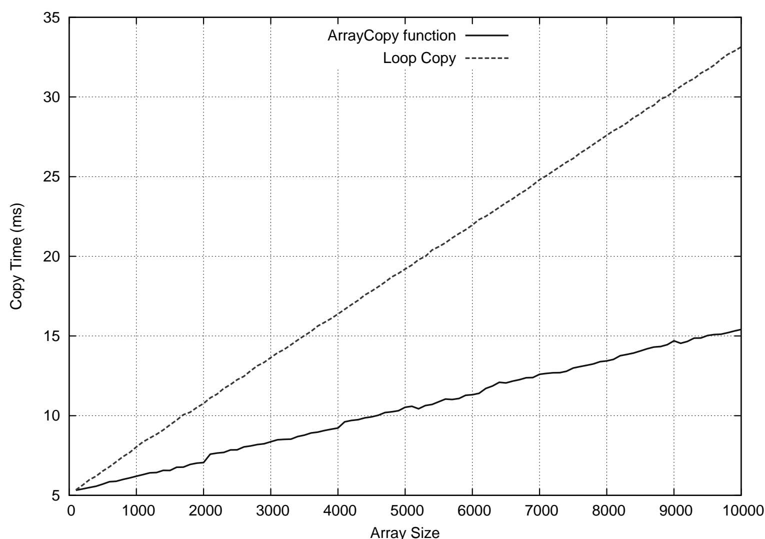


Fig. 4.7: Comparison of the System.arraycopy performance.

The various actions performed on these data structures, such as the swapping and replacing of array elements, can be a time-consuming task if each element has to be handled individually, e.g., copying elements using a `for` or `while` loop. However, the Java API provides a convenient `System.arraycopy` function, that may be used with appropriate pivot points within the array to extend, replace and re-order subsets of timing measurements within the array. The execution-time performance of this `System.arraycopy` function, running on a Sun SPOT mote, is illustrated in Figure 4.7. This figure shows double arrays of up to 10,000 randomly initialized

elements being copied using both the `System.arraycopy` function as well as a more traditional `for` loop iterating over each array element.

For very small arrays, the performance cost in copying an array is negligible, given that the execution time of the main software task is expected to be on the order of several hundreds of milliseconds. In addition, the `System.arraycopy` functionality will only be applied during the first few timing measurements. After the underlying timing behaviour has been sufficiently exposed through a larger number of measurements, no further significant alterations to the TimePredict arrays are then required unless the observed worst-case execution time is exceeded, or the system administrator decides to extend the ACETData array.

4.1.5 Accuracy and Clock Resolution

The accuracy of the timing estimates produced by TimePredict will be directly related to the accuracy of the timing measurement process (described in more detail in Section 4.2.1). The accuracy of each timing measurement will be constrained by the granularity of the system clock, which is set to 1ms for most standard Java-based applications. Any software task or function that executes, on average, in less than 20ms, introduces at unacceptably high clock error into the measurement process (i.e., 1 in 20, or 5%). Unless the clock resolution of the underlying system provides more accurate timing measurements, a more lengthy software function should be used as the basis for the timing measurements within TimePredict.

Luckily, the Sun SPOT mote provides two dedicated AT91 timer chips, that are available as part of the ARM processor architecture. The counters maintained within these timer chips are accessible though the Java Squawk APIs and allow time periods on the order of 1μsec to be measured [Chen et al., 2008]. However, although these timers offer a very high accuracy compared to the standard timestamp available in the `java.util.Date` library, the 16-bit counters used within the timer chip reset after a specified value has been reached (i.e., `0xFFFF` or $65,535$ iterations). Table 4.1, is taken from the Sun SPOT application notes [Goldman, 2007], and shows the minimum measurement interval, as well as the maximum duration of the various counters available via the AT91 timer/counter chips. For example, the MCK (Master ClocK) can be divided for the purposes of measuring set time periods, giving a possible clock resolution ranging from 0.0334μsec to 2.1368μsec. In contrast, the SLCK (SLow ClocK) provides a longer measurement interval of 30μsec, and greater maximum duration of up to 2 seconds. This 2

second upper limit is the largest timing interval that can be measured by the AT91 timer chips within the mote.

Clock	Clock Speed (KHz)	Time for One Tick (μsec)	Maximum Duration (ms)
MCK/2	29,952	0.0334	2.188
MCK/8	7,488	0.1335	8.752
MCK/32	1,872	0.5342	35.009
MCK/128	468	2.1368	140.034
SLCK	32.768	30.5176	2,000.0

Table 4.1: Available clock speeds and maximum durations on the Java Sun SPOT mote.

The actual class that performs the timing measurements within TimePredict is the `TimingEvent` class (see Figure 4.1). This class contains specific functions that are called at the beginning and again at the end of the target software function to measure its execution time. Since the execution time of a dynamically-adaptable system will be unknown initially, TimePredict uses the standard `java.util.Date` library to take millisecond-accurate timing measurements at first, since they have no set maximum duration. After an initial set of timing measurements have been generated, and the worst-case execution time shown to be much less than 2 seconds, timing measurements can revert to using the more accurate SLCK counter within the Sun SPOT. This allows time intervals to be measured to an accuracy of 30 microseconds (i.e., 30×10^{-6} seconds).

Occasional timing spikes can occur during execution, caused by a complex set of interactions between the software, hardware and operating environment. To reduce the possibility of any of these unusually large timing measurements exceeding the 2 second maximum duration for the SLCK, this high-accuracy measurement process is applied only to software with an average execution time of less than 500ms. Where the average execution time exceeds this threshold, the risk of exceeding the SLCK upper limit becomes too great. To mitigate this risk, the timing measurement process can revert to using the less accurate `java.util.Date` library. Although the inherent measurement error using this method is approximately 1 millisecond, the overall effect of this error on timing measurements in excess of 500ms is negligible.

4.1.6 Estimate Update Frequency

TimePredict, by default, is configured to automatically update both its average-case and worst-case timing estimates whenever new measurements become available. However, this initial update frequency may not be required at all times within all systems, e.g., within resource-constrained systems during periods of high load, or embedded devices in low-power mode (hibernation). Consequently, the implementation of TimePredict includes a number of possible update frequencies, that can be specified in advance by the system administrator, or automatically selected by TimePredict at run-time when the appropriate conditions arise. These update frequencies are,

- **Automatic Update**, the default update frequency, where each timing measurements is recorded and the timing estimates updated on every new measurement received.
- **Triggered Update**, although each timing measurement is recorded, the estimates are updated only when a measurement falls outside either the previous ACET or WCET bounds.
- **Active Sleep**, each timing measurement is recorded, but no updates to the timing estimates performed unless requested (e.g., by the Adaptation Manager within the system).

The limitations of the underlying hardware will be the main consideration when selecting an estimate update frequency. Even where there is sufficient processing and memory resources to generate regular timing updates, other considerations may dictate the estimate frequency, such as with battery-operated devices wishing to reduce power consumption. The principal hardware limitations, and their potential impact on the TimePredict approach must be assessed for each operating environment, to ensure that the functional and non-functional behaviour of the system is not degraded by an overzealous predictive process.

4.1.7 Impact on the System

The various calculations used within the timing analysis methods to fit and scale statistical distributions, to copy and sort arrays, and to keep track of the ongoing accuracy and precision of the various methods all consume processor cycles. Within the Java Sun SPOT, this leads to ‘normal’ functions being interleaved with timing analysis functions on the processor, resulting in a slight delay in the execution of the software compared to the same software deployed without TimePredict. One of the chief design requirements for the TimePredict approach is

to minimize any impact the on-going timing analysis would have on the normal execution of the dynamically-adaptable system. However, given that there is only one processor within the system, some analysis work will need to be performed at run-time, potentially at the expense of other processing.

As a means of minimizing the impact this analysis processing has on normal operations, the frequency of estimate update (previously discussed in Section 4.1.6), can be altered during periods of low load or timing volatility. In addition, the size of the worst-case and average-case arrays can be reduced when needed, to curtail the processing overhead required to iterate through each element. Any unnecessary consumption of system memory, caused by unused variables or recently dereferenced objects, is reduced by actively calling the garbage collection function, `System.gc()`, immediately following each refresh of the timing estimate. This is performed regularly since the availability of free memory within resource-constrained systems is often limited, and waiting for the JVM to initiate garbage collection may affect the timeliness of the software at a critical point. A comparison of the relative impact of this active garbage collection versus a JVM-initiated garbage collection is provided in the next chapter.

Lastly, the implementation of both the timing measurement process, as well as the predictive algorithms, are heavily optimized, to minimize the calculations required to produce a timing estimate. These software optimizations are described in more detail in the next section.

4.2 Software Implementation

Since the target hardware environment limits the choice of software to that supported by the Java Sun SPOT platform, TimePredict is implemented in a slightly restricted subset of the Java programming language, known as the Connected Limited Device Configuration v1.1 (CLDC) for Java Micro Edition (ME) [Smith, 2007]. This framework specifies a basic set of functions supporting a reduced set of APIs compared to the Java Standard Edition (SE). Specifically, common libraries such as those dealing with user interfaces and graphics are omitted, along with more specialized functionality such as reflection and user-defined class loaders. The latter two libraries are especially missed, since they play a key role in several Java-based dynamically-adaptable frameworks [Fritsch and Clarke, 2008] [Keeney and Cahill, 2003]. However, despite the lack of user-definable class loaders, the Sun SPOT motes and the Java ME APIs provide some very useful functions common to mobile phones and embedded PCs [Arseneau et al., 2006]

- over-the-air (OTA) software deployment, suites to encapsulate and compress standard Java class files as well as a remote O&M interface.

The functionality available within the Sun SPOT APIs permits a form of software adaptation, based on pushing new functionality to the mote from a server using an associated base-station. Due to the lack of reflection or bespoke class loaders within the Squawk JVM, modifications to the software must be composed remotely, compiled and then deployed to the mote with the assistance of an external Adaptation Server, using the available OTA deployment functionality. Although this restricts software adaptation to a form of on-request push deployment, the software running on the mote may still be considered as being dynamically-adaptable, since the determination of when and what changes to make to the system are performed locally and relayed to the Adaptation Server for implementation.

This push adaptation requires the Adaptation Server to automatically compose, compile and deploy Java class files, as well as maintain a set of alternate functional components to be deployed when requested by the mote. For the purposes of this work, a set of inter-changeable configurations of the software are used to modify the functionality (and timing behaviour) of the Java Sun SPOT mote during run-time. Although the Squawk JVM supports a type of re-deployable Java thread known as an `Isolate`, the various configurations of the Sun SPOT software composed by the Adaptation Server are compiled as `MIDlets`, formed from a suite of compressed Java class files.

4.2.1 Timing Measurement

The actual timing measurement process underlying the entire approach is provided through dedicated timing hardware on the Sun SPOT motes, using sub-millisecond accurate time-stamping. Where the expected timing behaviour of the software is very large (i.e., several seconds in duration), this time-stamping is performed using the `java.util.Date` library. Although the target software being measured can be as fine-grained as a single line of code, it is more typical (and informative) to evaluate a more detailed software function, that can provide a better representation of the timeliness of the software as a whole.

The `TimingListener` interface, described in Figure 4.8, outlines two functions to measure the execution time of an identified code-segment, function or class within the target software. The `timerStart()` and `timerStop()` functions are used as time-stamps, with the `TimingEvent`

class providing functionality to measure the intervening period between the initial and final time-stamp.

```
public interface TimingListener
{
        public void timerStart(TimingEvent event);
        public void timerStop(TimingEvent event);
}
```

Fig. 4.8: TimingListener interface.

The standard Java timing measurement functionality uses the `java.util.Date` library, which can return the number of milliseconds elapsed since mid-night on the 1st January 1970. A `Date` object can be created as a time-stamp either side of a target software function, and the elapsed time calculated by finding the difference between each time-stamp. Calls to the `getTime()` function within the `Date` object return a `long` value giving the elapsed time. More accurate timing measurements can be created using the dedicated timing hardware on the Sun SPOT mote. The function calls used to initialize and use AT91 timer are illustrated in Figure 4.9. A `counter` is used to measure the number of ticks recorded by the timer, and the elapsed time can then be calculated by multiplying this value by the time of each tick. The function being measured is encapsulated by two instructions to start and to stop the timer. When the elapsed time is calculated, the value must be divided by $1,000$ to provide the time in milliseconds rather than microseconds (μSecs).

4.2.1.1 Adding Timing Measurements

The average-case data structure is a simple one-dimensional array, and is used to store each new timing measurement produced within the system. However, the array has a pre-determined capacity, to limit the memory overhead and computational workload placed on the system. Once the ACET array has been filled, each new timing measurement added to the array progressively replaces the oldest remaining measurement. The capacity of this array is determined by the number of measurements required to fit the smoothing parameters for the ES model. These

```
int counter;
IAT91_TC timer = Spot.getInstance().getAT91_TC(0);

//Set the slow clock
timer.configure(TimerCounterBits.TC_CAPT | TimerCounterBits.TC_CLKS_SLCK);
timer.enableAndReset();         //Start the timer

//Execute the target software function
myFunction();

counter = timer.counter();      //Find the elapsed time
timer.disable();                //Stop the timer

//Calculate the time in milliseconds
double timeInterval = ((counter * 30.5176)/1000.0);
```

Fig. 4.9: Using the AT91 timing functionality.

smoothing parameters are fitted by generating estimates for the previous n measurements, and measuring the divergence of this estimate from the subsequent timing performance of the software, i.e., the estimation error. A good fit for the ES model parameters would be values that minimize this error, and offer the best possibility of providing an accurate estimate of likely future timing behaviour. Conversely, the more measurements that are included in this analysis, the greater the processing and memory usage on the underlying system. Relatively shorter lag lengths introduce fewer terms into the resulting timing analysis and require less processing, but can result in less accurate timing estimates. The default setting for the lag length within TimePredict is 50, so the ACET array maintains an array of the same size. Timing data is added to the array in order of occurrence, since the ES average-case model accords more relevance to more recent timing data when producing a bounded ACET estimate. Each timing measurement is recorded as a double value, representing the timing in milliseconds, regardless of the clock resolution of the underlying system. Algorithm 1 describes the addition of new timing measurements (t) to the ACET array, with the index of the oldest element within the array (i) used to determine the element to be replaced with the next timing measurement. Initially, the size and index of the array are set to zero, and the capacity to the default lag length of 50.

The worst-case data structure is slightly different, in that there is an emphasis on more

Algorithm 1 Adding timing measurements to ACET array

Require: $t \geq 0$, and initially that $size = 0$, $capacity > 0$, $i = 0$

Ensure: Array retains order-of-occurrence, from oldest (i) to newest $(i - 1)$

```
if (size < capacity) then
    array[i] := t
    if ((i + 1) = capacity) then
        i := 0, size := capacity
    else
        i := i + 1, size := size + 1
else
    array[i] := t
    if ((i + 1) = capacity) then
        i := 0
    else
        i := i + 1
```

extreme timing measurements over more recent observations of the system. Algorithm 2 describes the addition of worst-case timing measurements to this data structure within the `WCETData` class. An integer array is used to create a probability density function (in effect a frequency distribution) of the timing measurements recorded thus far, over a restricted range of values. Whereas the size of the array represents the range of timing measurements recorded within the system, each integer element within the array represents the number of times a particular measurement, or range of measurements, have been observed. Since only the most extreme timing measurements are useful within any worst-case analysis, and since the observed worst-case timing values may change during execution, the WCET array can dynamically shift elements when more extreme timing values are encountered. The default array size is 100, with an initial interval of 1ms, so that each element within the array represents a single millisecond leading up to the observed maximum execution time. In effect, the bounds on the array size are set by the range of potential timing measurements encountered during execution, with the initial assumption being that the worst-case timing behaviour will be clustered within a 100ms interval. If this is insufficient, the interval for each element within the array can be increased, to encompass a greater range of timing measurements.

Algorithm 2 Adding timing measurements to WCET array

Require: $t \geq 0$, and initially that $size = 0$ and $capacity > 0$

Ensure: Array records the frequency of values from $t_{max-size}$ to t_{max}

```
if (t ≥ min and t ≤ max) then
    i := getIndex(t)
    array[i] := array[i] + 1
else
    if (t > max) then
        max := t
        min := (t − max) + min
        if (|max − min| < capacity) then
            resizeArray(max − min)
        else
            shiftArrayRight(t − max)
        array[i_max] := 1
```

4.2.2 Algorithm Optimizations

The predictive algorithms used within TimePredict can prove computationally intensive for resource-constrained devices if not implemented in an optimal manner. These algorithms, especially within the `GEVModel` and `ESModel` classes, use recursive functions to efficiently fit and test the various statistical distributions (and smoothing parameters) to model the underlying data. For the GEV models that contain a number of unknown parameters, recursive functions provide a divide-and-conquer approach towards the estimating the distribution parameters that provide the best fit for the underlying timing data.

The initial step for fitting the worst-case models within the `GEVModel` class, is to scale the GEV/GDP distribution until it matches the cumulative frequency distribution of the data. For all distributions except the Gumbel distribution, a further step is required, to adjust an additional parameter (α) to find the best fitting shape for the distribution. A recursive function, called **fitAlpha**, is used to progressively narrow the likely range of the shape parameter, and is illustrated in Algorithm 3. The **getFit** function described in Algorithm 3 is the fitness function for the distribution, and returns a numeric value that converges to 0 the closer the distribution fits the data. The estimated value for alpha ($\hat{\alpha}$) is the result of fitting the distribution using the initial range described from α_{min} to α_{max}. This initial range is progressively halved until it is less than a pre-defined initial value specified by α_{inc}. The closer this α_{inc} value is to 0 the more

Algorithm 3 Recursive parameter estimation for worst-case statistical models.

Require: $0 \leq \hat{\alpha} \leq \alpha_{max}$, and $\alpha_{inc} > 0.0$

Ensure: $\hat{\alpha}$ provides the best fit for the available data.

```
if (|α_max − α_min| ≤ α_inc) then
    α̂ := (α_max − α_min)/2
    return (α̂)
else
    α_low := (α_min + α̂)/2
    α_high := (α̂ + αmax)/2
    //Assumes the getFit function returns a numeric value
    if getFit(α_low) < getFit(α_high) then
        //Recursively call the function again with lower bounds
        return fitAlpha(α_min, α̂)
    else
        //Recursively call the function again with higher bounds
        return fitAlpha(α̂, αmax)
```

iterations of the recursive **fitAlpha** function are required to find the best fit for the $\hat{\alpha}$ parameter.

Similarly, the smoothing parameters associated with the ES model must be continuously refitted at run-time to ensure the resulting average-case estimate closely matches the actual timing behaviour of the software. Similar to the algorithm used in the `GEVModel` class, a recursive algorithm is used within the `ESModel` class to find a close approximation of the optimal smoothing parameters α and β for the predictive model (see Equation 3.5 in the previous chapter).

Algorithm 4 Recursive estimation function for the ES model smoothing parameters.

Require: $0 \leq \alpha_{min} \leq \alpha_{max}$, and $\alpha_{inc} > 0.0$

Ensure: $(\alpha_{max} + \alpha_{min})/2$ provides the best approximation for the smoothing parameter α.

```
while (α_max − αmin > α_inc) do
    pivot = (α_max + α_min)/2
    α1 = (pivot+α_min)/2
    α2 = (pivot+α_max)/2
    if (getSumSquares(α1) < getSumSquares(α2) ) then
        α_max = pivot
        fitAlpha(α_max, α_min)
    else
        α_min = pivot
        fitAlpha(α_max, α_min)
```

Algorithm 4 describes the operation of the recursive `fitAlpha` function within the `ESModel` class. This function starts with a maximum and minimum range for the parameter being estimated, in this case α, and recursively reduces this range until it is less than or equal to a threshold value specified by α_{inc}. A similar function is also included in the `ESModel` class, with both calculating the sum of the squared difference between previous measurements, and iterative estimates taken from various ES models defined by α_1 and α_2.

4.2.3 Model Validation

A large number of subtle factors can affect the timing behaviour of software, including system load levels, resource contention issues, user interrupts, hardware or software errors, CPU caching policies, CPU branch prediction, memory availability as well as the software execution history. The complex interaction between these factors can result in seemingly random perturbations in the timing behaviour of software, and can occasionally induce timing delays in even very basic software functions. Assessing the predictive accuracy and precision of the various forecasting methods used within TimePredict requires on-going testing, both to avoid introducing unwanted errors into the codebase, as well as validating the functionality of the existing software. To support the implementation of the various predictive models, a test framework is required that can generate representative time series data (e.g., the execution times of different types of software functions), that can in turn be used to test the performance of the predictive models used in TimePredict.

Typically, measurement-based timing analysis approaches validate their predictive models against basic software functions executed under ideal circumstances [Edgar, 2002] [Hansen et al., 2009]. However, the changeability inherent within dynamically adaptable software systems would be difficult to re-create, unless the software contained an inherent variability that approximated the type of changes in timing behaviour likely to occur within dynamically-adaptable systems. A series of representative measurements, using time series data similar to what would be expected within the execution times of dynamically-adaptable systems, would provide an appropriate test framework for the models. Although the worst-case and average-case models used within TimePredict have been designed for run-time operation, a series of previously recorded timing measurements could provide a simulated run-time environment, if applied in the same order as they were recorded. In this way, a test framework based on previously recorded time series data

offers a limitless number of 'run-time' timing measurements, and allows testing against software timing behaviours that would be difficult to re-create through the measurement of a live system. By iterating through the measurements off-line, the performance of each of the various predictive models can be examined in greater detail than what would be possible executing on a live system. Also, since the test timing measurements are generated in advance, and tested afterwards without the real-time restrictions of a live system, the implementation of the TimePredict approach can be assessed independently of the underlying hardware, with a user-defined interval between consecutive timing measurements that permits debugging and code analysis.

The software timing measurements used to guide the implementation of TimePredict are taken from the recorded execution times of a software benchmark suite, running on a Java Sun SPOT mote. The benchmarks used include some standard functions, such as matrix multiplication [Arndt et al., 2009] and prime number evaluation, as well as some non-standard benchmarks used to assess the performance of the I/O functions, memory allocation and garbage collection within an embedded system.

- A matrix multiplication benchmark.
- Prime number benchmark.
- I/O benchmark function.
- Memory assignment benchmark.

The matrix multiplication benchmark multiplies two square matrices of a specified size, with matrices of size n requiring n^2 individual multiplication operations. The prime number benchmark counts all the prime numbers that occur between 1 and user-defined upper limit, with progressively larger upper limits resulting in a linear increase in its execution time. The I/O benchmark function iterates through each of the I/O pins and on-board sensor boards present on the Sun SPOT, recording the changes that occur within each. This benchmark evaluates the speed of the communications bus within the Sun SPOT mote, as well as performance of the various I/O interrupts used by the system to poll the on-board sensors. By changing the number of pins/boards polled, the execution time of the benchmark (and the impact on the underlying system) can be likewise altered. The final benchmark function used is a memory assignment benchmark, that declares an array of double values within memory, populates this array with

random values, and then re-sizes this array to remove all the elements. This benchmark tests the speed and capacity of the memory resources within the Sun SPOT mote, as well as the garbage collection function within the JVM that cleans up the deallocated array data after each iteration. The memory assignment function requires approximately n operations (random value assignments), where n is the size of the array.

By varying the inputs into these benchmark functions, their resulting execution times within the Sun SPOT mote can be made to mimic the likely timing behaviour of dynamically-adaptable software. Typically this timing behaviour consists of extended periods of little variation around a specific average value, interspersed with sudden changes that increase timing variation and/or alter the average execution time of the software.

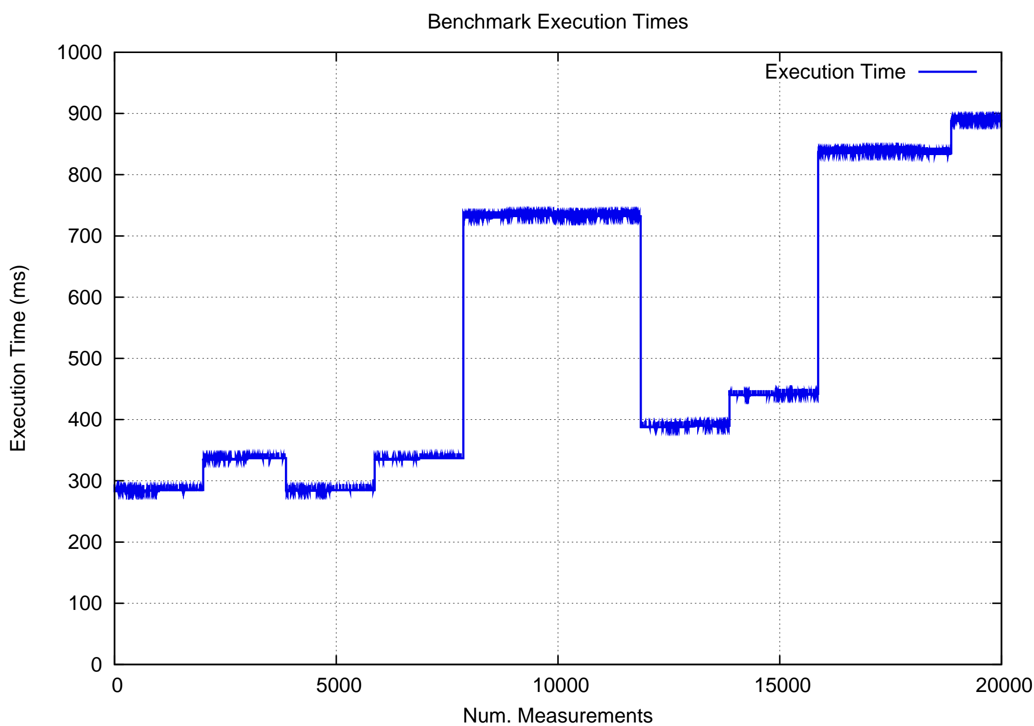


Fig. 4.10: Execution times for the combined software benchmark functions, over 20,000 measurements.

For testing purposes, the benchmark functions execute within a particular configuration for 1,000 iterations, after which their input parameters are randomly altered to values within a

pre-defined range. This in turn will change the timing behaviour of the benchmark suite every 1,000 measurements to some previously unseen timing behaviours, mirroring the likely changes that may occur at run-time within dynamically-adaptable systems. In total, the benchmark functions were executed over 20 individual configurations, with each configuration comprising 1,000 timing measurements, giving a total validation dataset of 20,000 timing measurements. This timing behaviour is illustrated in Figure 4.10, and provides a comparable dataset of time series values that could be expected within a live operating environment.

The 20 configurations used during the execution of the benchmark suite are apparent within Figure 4.10, where the recorded execution times show changes at regular intervals of 1,000 timing measurements. Using this time series data as the basis for the implementation testing of TimePredict, will follow the same kind of performance associated with unanticipated functional adaptations occurring within the system at run-time. The evaluation of the accuracy and precision of TimePredict's estimates, when applied to this dataset, is presented in the next chapter, however, the analysis of the timing measurements themselves is presented in Table 4.2. This table shows the break-down of the benchmark time series data both individually, and collectively over the course of 20,000 measurements. What is immediately apparent is the compact distribution of the measurements, as indicated by their low inter quartile range (IQR). The inter-quartile range is a measure of statistical dispersion, calculated as the middle 50% of the data found between the first and third quartiles, i.e., between the bounds the encompass 25% and 75% of the dataset.

Having a low IQR indicates the variation within the timing measurements is relatively small, and clustered around a mean value. However, the larger absolute ranges (i.e., the difference between the recorded maximum and minimum) suggests that the measurements within a single configuration includes occasional values that are significantly different from the mean. Within the Sun SPOT motes, occasional calls to the garbage collection function made automatically by the JVM may be responsible for timing spikes. The garbage collection function is a function that de-allocates memory structures, and variable no longer in use by the system, and can be called by the user (e.g., `System.gc()`), or more typically is invoked by the JVM. Similarly, other high-priority system tasks related to hardware interrupts may occasionally interfere with more normal processing on the CPU, and bring about similar jumps in timing behaviour. Other, more systematic 'steps' or 'jumps' in the data occur when the benchmark suite is reconfigured after 1,000 iterations. These can be identified as they occur at more regular intervals and are

illustrated in the range of the box plots within Figure 4.11.

Config.	Min (ms)	Mean (ms)	Max (ms)	Range (ms)	Q_1	Q_3	IQR
1	283	283.727	300	17	283	284	1
2	284	285.192	336	52	285	285	0
3	335	335.213	345	10	335	335	0
4	283	330.172	346	63	337	337	0
5	283	284.028	294	11	284	284	0
6	285	291.778	343	58	285	285	0
7	335	335.372	345	10	335	335	0
8	337	389.492	741	404	337	337	0
9	728	730.749	743	15	730	731	1
10	731	733.498	746	15	733	734	1
11	730	731.812	743	13	731	732	1
12	388	687.318	745	357	732	733	1
13	387	388.348	398	11	388	388	0
14	389	396.512	448	59	389	390	1
15	439	440.309	452	13	440	440	0
16	441	494.453	847	406	442	442	0
17	834	836.534	848	14	836	837	1
18	835	837.921	852	17	837	838	1
19	833	842.314	898	65	834	836	2
20	887	888.786	900	13	888	889	1
Overall	502.35	527.1764	583.5	81.15	523.05	523.6	0.55

Table 4.2: Execution times of the software benchmark suite, over 20 configurations.

The box plots in Figure 4.11 describe the same summary data presented in Table 4.2. The black lines within the data represent the means, with the larger thin lines describing the range (minimum and maximum values) for each configuration within the dataset. Although non-existent or too small to be included in this figure are the IQR values, which are generally within one millisecond either side of the mean value. By comparing the box plots to Figure 4.10

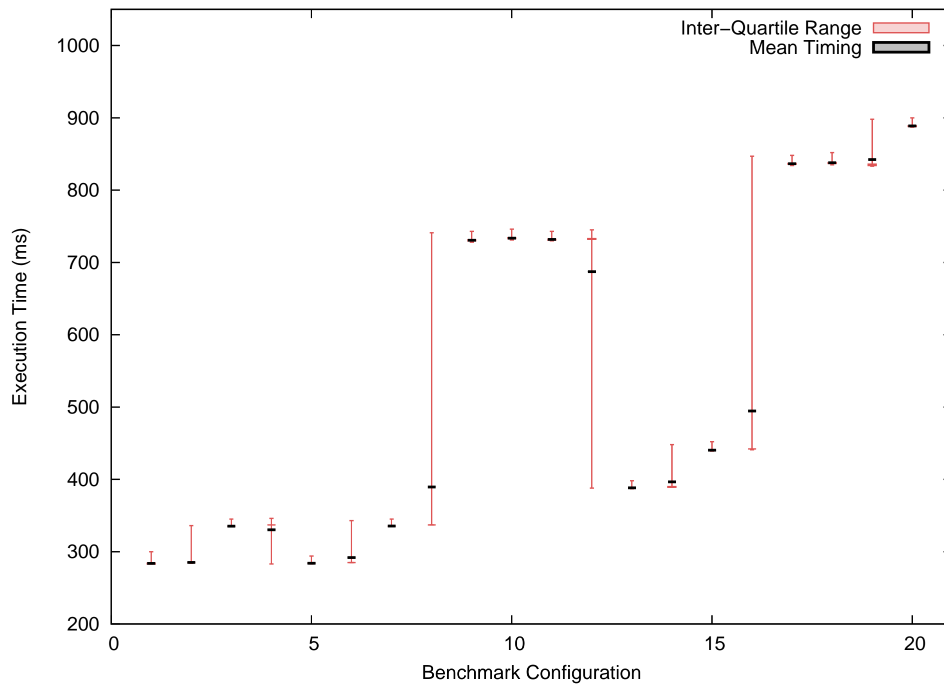


Fig. 4.11: Boxplot showing the measurements taken of the software benchmark functions, across each of the 20 recorded configurations.

illustrated previously, we can see that the large range of Configurations 8, 12 and 16 can be ascribed to the dramatic change in timing behaviour from the previous benchmark configuration.

In summary, the off-line time series data recorded from the execution time of four benchmark functions provides a useful analog for the type of timing behaviour likely to be experienced at run-time within a dynamically-adaptable embedded system. In the next chapter, the evaluation of this dataset is carried out in order to demonstrate the applicability of TimePredict's statistical methods as forecasting tools, albeit within a virtual operating environment where resource availability issues, user interrupts or fluctuating system loads cannot occur. Consequently, an additional assessment is required to fully evaluate TimePredict, so that its performance can be examined within a live operating environment subject to system resource constraints and other run-time considerations. The application scenario used to provide the basis for this run-time analysis is described in the next section, whereas the evaluation of TimePredict using both the benchmark data, and the run-time data, is presented in the next chapter.

4.3 Dynamically-Adaptable Application Scenario

Wireless Sensor Networks (WSNs) are typically composed of a large number of low-cost wirelessly-enabled sensor devices, that collaborate in order to measure a specific set of physical or environmental conditions. The sensor motes can be deployed to a particular environment in an unplanned manner (i.e., scattered over an area), relying instead on the self-organizing capabilities of each device to establish a radio network, and begin taking and then transmitting sensor data. However, sensor devices are prone to failure, usually due to the limitations on their power supply. To cope with unexpected device failures, as well as any resultant changes to the network topology, each device commonly employs broadcast communication to propagate its sensor data throughout the network. However, rather than flooding the network with raw sensor readings, each device may first perform some local computation on the collected sensor data before transmitting a fully or partially processed version of the required information [Akyildiz et al., 2002].

The volatile operating environments typically encountered by WSNs require each sensor mote to be highly autonomous, capable of executing without supervision for extended periods, and be able to automatically optimize its behaviour to suit the prevailing operating conditions. When deployed within harsh or dangerous environments, the ability to reconfigure the mote software at run-time, as well as remotely deploy new functionality when needed, avoids the difficult task of gaining physical access to the device in order to implement the desired software changes [Barrenetxea et al., 2008]. However, adaptations to the functionality on the sensor mote must be configured to optimize its performance for the current local operating conditions, without overly taxing the limited resources on the device, or degrading its quality of service in reporting specific environmental conditions. As more potential software configurations are made available, and the suitability of each configuration becomes limited to a very specific set of operating conditions, the optimal configuration for the system may only be selected at run-time in response to changes detected in the operating environment. To maintain a previously agreed QoS, each sensor mote must continuously evaluate its performance, including its timing behaviour, to identify the point at which functional adaptations may be required to optimize the system.

In order to illustrate the performance of TimePredict, as well as emphasize its suitability for use within resource-constrained devices, the next section describes the analysis-driven reconfiguration of a dynamically-adaptable wireless sensor mote. Using the run-time estimates

provided by TimePredict, the dynamically-adaptable sensor device can select a software configuration so as to maintain a specified QoS, despite unanticipated variation within its operating environment. Since the deployment of a large-scale WSN is impractical as well as unnecessary in order to illustrate the performance of TimePredict, the application scenario described in the next section presents a small-scale WSN sufficient to evaluate the accuracy, precision and system impact of TimePredict on a single device. Although this scenario only considers the timeliness of dynamically-adaptable software running on a single sensor node, it is envisaged that in a real-world environment, many dynamically-adaptable motes are available as part of a larger WSN, each adapting its functionality to exploit changes in the local operating conditions.

4.3.1 Experimental Setup

The small-scale WSN used to evaluate the performance of TimePredict consists of two nodes, a Java Sun SPOT device that fulfills the role of a dynamically-adaptable sensor mote, and a network-connected Adaptation Server that provides reconfiguration support and logging functionality. As illustrated in Figure 4.12, TimePredict is deployed on the sensor mote, and used to furnish timing estimates as inputs into an on-going software evaluation and adaptation process. This adaptation process is controlled by a dedicated Adaptation Manager function executing on the sensor mote. When the execution time of the software is deemed to have exceeded a specified threshold, the Adaptation Manager initiates an adaptation action to replace the current configuration of the system with a more suitable alternative. However, since neither the standard Java reflection APIs nor any user-definable class loaders are supported by the JVM on the mote, software adaptations must be pushed to the device using the Over-The-Air (OTA) deployment functionality available in the Sun SPOT SDK. An Adaptation Server is used to deploy new software to the Sun SPOT, in addition to providing a server-side configuration repository, and remote logging functionality. While the actual OTA deployment is performed by the Adaptation Server, the initiation and selection of adaptations is controlled by the Adaptation Manager on the mote. To avoid unnecessary memory usage logging performance data within the mote, the timing estimates generated by TimePredict are recorded on the Adaptation Server, and stored for later off-line evaluation of TimePredict's performance. Wireless communication between the Sun SPOT sensor and the Adaptation Server is supported through ZigBee (as described earlier in Section 4.1.2.1). Both sensor data and logging information are broadcast using a buffered

data-stream connection to ensure the correct in-order delivery of messages to the server.

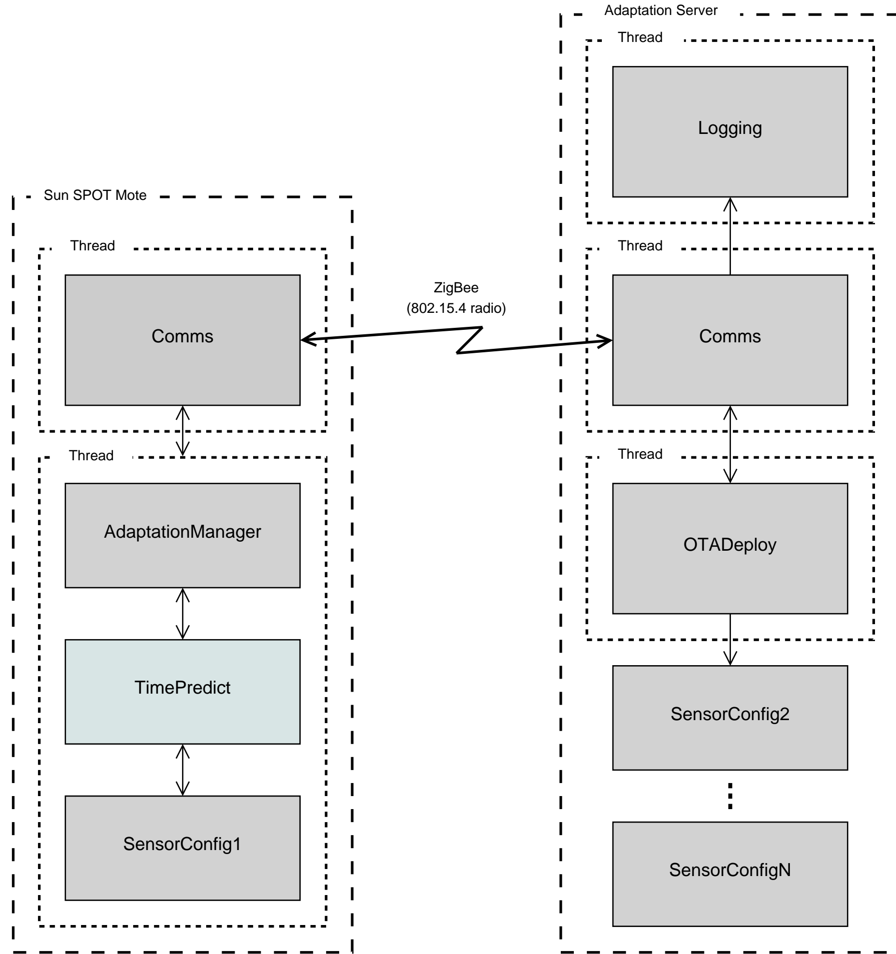


Fig. 4.12: Experimental setup with Sun SPOT mote and Adaptation Server.

The Sun SPOT mote employed as the wireless sensor device uses the latest version of the Sun SPOT SDK (v5.0 RED). The application development environment for the Sun SPOT motes (and the Adaptation Server) is provided by Java NetBeans version 6.5.1, running on a Linux-based PC, and enabled with the Java Sun SPOT extensions. The Adaptation Server is installed on a Dell Inspiron 1750 laptop, running Ubuntu Linux 10.04 (Kernel version 2.6.32-24), communicating with the Sun SPOT mote via a USB-connected Sun SPOT base-station. The application running on the Adaptation Server is based on Java Standard Edition version 1.6.0. To facilitate the run-time deployment of new software configurations to the mote, a system call

is made from inside the `OTADeploy` class running on the Adaptation Server to an Ant build environment, co-located on the Adaptation Server. Apache Ant is a Java-based build tool, that is provided by the Sun SPOT SDK to enable OTA software deployment. The version of Ant running on the Adaptation Server is v1.7.1.

The light, temperature, tilt and acceleration sensors available within the Sun SPOT mote provide sensor information that can be processed at run-time, and a summary of the collected data then transmitted to the Adaptation Server. The amount of pre-processing performed, as well as the size of the collected sensor data, largely determines the timeliness of the application running on the mote. Where there is little variation within the sensor readings, the mote may conserve power by transmitting only highly summarized sensor data, accumulated over a longer period, and analyzed entirely on the mote itself. However, within highly variable operating environments, a more efficient strategy may be to rapidly broadcast sensor information, while scaling back the amount of pre-processing performed locally. The time it takes to execute this sensor analysis function can provide an indication of the suitability of the current configuration of the software for the prevailing operating conditions. A lengthy analysis process executing within a highly-variable environment may be too slow to react to important changes within the sensor data. Conversely, a relatively faster analysis process, executed repeatedly within a stable operating environment, can needlessly consume power and system resources reporting unchanged sensor information.

4.3.2 Experimental Goals

The experimental evaluation of TimePredict uses this dynamically-adaptable sensor application to estimate the timeliness of software running on a resource-constrained device that is subject to modification at unanticipated periods during run-time. The principal goals of this experiment are;

1. To illustrate the accuracy and precision of TimePredict in forecasting the execution time of dynamically-adaptable software.
2. To perform this analysis at run-time on a resource-constrained device.
3. To show that the timing analysis process can be executed without adversely affecting the underlying system.

Although the requirement for dynamic software adaptation is grounded in the specialized nature of the TimePredict approach, the motivation driving dynamic software adaptation, as presented in this application scenario, is secondary to the evaluation of the performance of TimePredict. The impetus behind using a dynamically-adaptable sensor application is to provide a challenging operating environment to highlight the performance and operational characteristics of the timing analysis process, rather than offer an ideal use-case for dynamic software adaptation.

The execution time of the sensor processing function on the Sun SPOT mote provides the timing measurements used by TimePredict. By recording these timing measurements, as well as the estimates produced from these measurements, the run-time accuracy and precision of TimePredict can be assessed. To determine the impact of TimePredict on the normal operation of the software, as well as the added workload it places on the underlying system, a comparison is performed between a configuration of the software deployed with TimePredict, and the same configuration deployed without timing analysis functionality. The system impact due to TimePredict can be evaluated by measuring the difference in software timeliness, memory usage and power consumption between these two configurations of the sensor software.

In order to evaluate the practicality of using timing estimates as the basis for an adaptation process, a number of different configurations of the mote software are described in the next section. Each configuration performs a slightly different sensor analysis, so that each configuration will have a different expected execution time. By comparing the timeliness of the current software configuration, to an estimate of its worst-case and average-case behavior, a determination can be made to adapt the system for a more computationally intensive configuration, or a faster configuration with less processing overhead.

4.3.3 Application Configurations

Table 4.3 presents sixteen distinct configurations of a dynamically-adaptable sensor mote, with each configuration of the mote software analysing different sensors, applying different analysis methods, and evaluating a different number of sensor measurements. The configurations are ranked approximately in the order of their computational complexity, with Configuration 1 offering the most straight-forward sensor analysis, and Configuration 16 the most potentially demanding. The table describes the four separate sensors available on the Sun SPOT mote, namely the light level, temperature (temp.), tilt and acceleration (accel.) sensors. Configurations 1 to 8

use only the light and tilt sensors, whereas the remaining configurations include all four sensors in the analysis process.

Config. No.	Sensors Deployed				Sensor Analysis Performed	Sample Size
	Light	Temp.	Tilt	Accel.		
1	✓		✓		Mean	100
2	✓		✓		Mean & Median	100
3	✓		✓		Mean, Median & Std. Dev.	100
4	✓		✓		Mean, Median, Std. Dev. & Correlation	100
5	✓		✓		Mean	1000
6	✓		✓		Mean & Median	1000
7	✓		✓		Mean, Median & Std. Dev.	1000
8	✓		✓		Mean, Median, Std. Dev. & Correlation	1000
9	✓	✓	✓	✓	Mean	100
10	✓	✓	✓	✓	Mean & Median	100
11	✓	✓	✓	✓	Mean, Median & Std. Dev.	100
12	✓	✓	✓	✓	Mean, Median, Std. Dev. & Correlation	100
13	✓	✓	✓	✓	Mean	1000
14	✓	✓	✓	✓	Mean & Median	1000
15	✓	✓	✓	✓	Mean, Median & Std. Dev.	1000
16	✓	✓	✓	✓	Mean, Median, Std. Dev. & Correlation	1000

Table 4.3: Configurations of the sensor software.

The various readings taken from the sensors are stored in the Sun SPOT memory, with the sample size (illustrated in Table 4.3) denoting the number of measurements recorded and used in the sensor analysis. The larger the sample size, the more sensor measurements that must be evaluated, and the longer the likely execution time of the analysis function. For the purposes of this application, two sample sizes are arbitrarily selected, 100 and 1000, to illustrate the effects of an increased processing overhead on the expected execution time. In addition to varying the sensors used, as well as the number of measurements collected, different analysis methods are

applied to the data to further differentiate the processing overhead (and execution time) of the various configurations.

The basic configuration of the sensor analysis function consists of a calculation of the mean value for the collected sensor measurements. The various other software configurations augment this by evaluating additional statistical properties of the sensor data, including the median sensor value, the standard deviation of the collected measurements, and correlation of the various sensor readings. The first three statistical properties are relatively straight-forward, with the fourth, the correlation function, used to evaluate how the variation within the value of one sensor mirrors variation in others. For four sensors (i.e., Configurations 9 to 16) a total of six correlation values are derived, i.e.,

- Light vs. Temperature
- Light vs. Tilt
- Acceleration vs. Tilt
- Acceleration vs. Light
- Temperature vs. Acceleration
- Tilt vs. Temperature

The correlation between the tilt and the acceleration of the mote is omitted, since the two sensor readings are not independent, i.e., a change in tilt will register as a similar change in the acceleration of the mote.

4.3.3.1 Normal Operation

Each software configuration will need to be executed multiple times both to provide a set of timing measurements for TimePredict, as well as to evaluation the impact of the timing analysis on the underlying system. The expected timeliness of the software is principal driver for the adaptation process. However, the subtle factors that influence software timeliness are not easily anticipated, e.g., memory usage, concurrency issues and garbage collection, with a result that the interval between consecutive adaptations cannot be determined with any great accuracy. For the purposes of this experiment, and to ensure that adaptations take place, the timing threshold used by the Adaptation Manager can be set to an artificial level after a specified number of iterations, to ensure an adaptation is triggered, and a new configuration of the system deployed.

For each configuration of the sensor analysis software, the Adaptation Manager maintains a high and low timing threshold. The high timing threshold is used to indicate when the next

(more computationally intensive) configuration of the software may be safely deployed. The low timing threshold denotes the minimum required execution time for the currently deployed software, signalling to the Adaptation Manager when a less processor intensive configuration should be introduced. The timing estimates provided by TimePredict are input into the Adaptation Manager, and compared against these threshold values to initiate a particular adaptation action.

The adaptation plan for the dynamically-adaptable sensor mote starts with the mote deployed and executing Configuration 1 of the software. After a set number of iterations of each configuration, the Adaptation Manager progressively adapts the software through each configuration in turn, until Configuration 16 is reached. This adaptation plan represents the behaviour of a dynamically-adaptable sensor mote operating in a progressively more stable operating environment, i.e., continually updating its functionality to provide further analysis of the available sensor information. As the adaptations introduce more computationally demanding configurations, the timing behaviour of consecutive configurations of the software should be markedly different, and suited towards the reactive run-time predictive models found within the TimePredict approach. The execution of this adaptation plan can be summarized as,

1. The mote is initially deployed with Configuration 1 of the software, including the TimePredict functionality.

2. A discovery protocol is first executed to find the Adaptation Server and open a connection to transmit sensor analysis and logging data.

3. The mote begins collecting sensor measurements, performing an analysis on these readings and broadcasting the results to the Adaptation Manager.

4. After each call to the sensor analysis function, its execution time is measured and TimePredict updated. The timing estimates for the software are refreshed, and forwarded to the Adaptation Manager as well as broadcast to the server for logging.

5. After a specified number of iterations, the Adaptation Manager adjusts its timing threshold so that the TimePredict estimates trigger an adaptation action. Alternatively, the Adaptation Manager can continue execution until the execution time of the software exceeds this threshold by itself, however the interval between adaptation actions being called would be unknown.

6. An adaptation request is transmitted to the Adaptation Server, which deploys the required configuration of the software, and restarts the mote. The discovery protocol re-establishes communication, and the execution continues with an updated sensor analysis function.

The processing and timing analysis periods for the various configurations of the software running on the mote are illustrated in Figure 4.13. Once the sensor data has been processed, a response is sent to the Adaptation Server, and a fresh timing measurement is added to TimePredict. The WCET and ACET bounds are then re-calculated, and forwarded to the Adaptation Manager within the Sun SPOT, and transmitted as log data to be stored on the Adaptation Server.

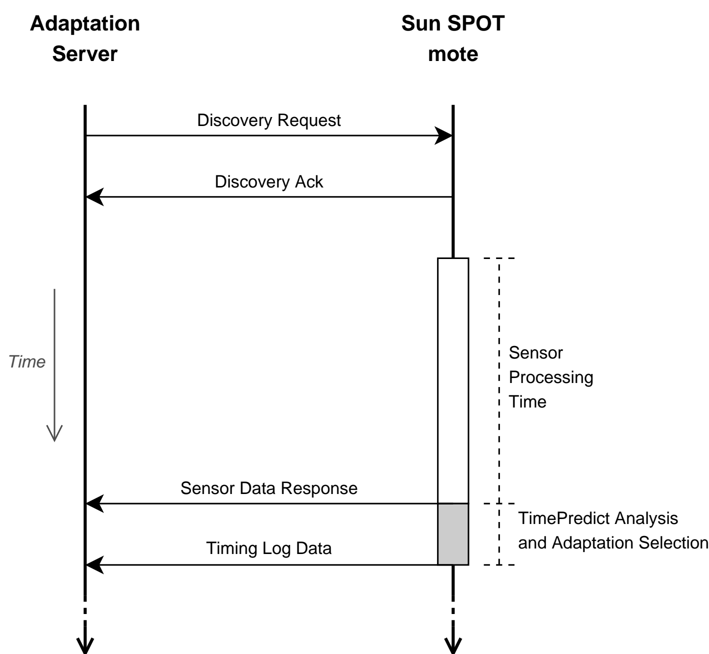


Fig. 4.13: Execution order during normal operation.

Assuming the updated timing estimates are within a pre-defined threshold, e.g., the WCET bound is less that the threshold value maintained within the Adaptation Manager, the configuration of the software remains the same. However, if the Adaptation Manager determines the functionality currently deployed on the mote is non-optimal, an adaptation request is then made to the Adaptation Server. Figure 4.14 illustrates the initiation of an adaptation request from the mote, and the deployment of a new configuration of the software from the Adaptation Server.

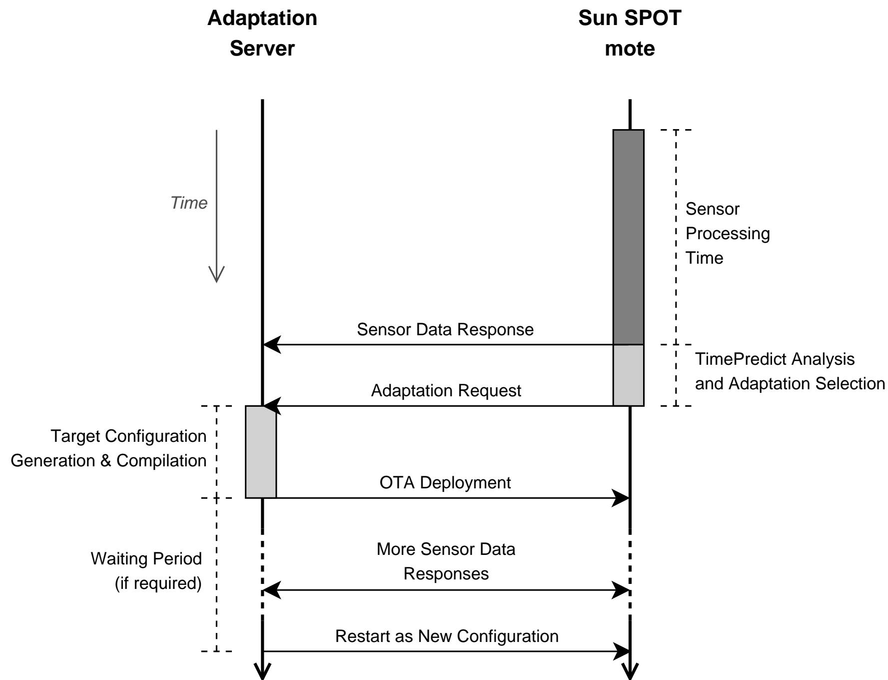


Fig. 4.14: Adaptation request and software deployment.

The Adaptation Server must compose and compile the various Java files locally, and then deploy the resulting bytecode `MIDlet` to the mote, using the OTA deployment functionality within the Sun SPOT SDK. Up to eight separate `MIDlet`s can be deployed to the mote, however only one is executing at any one time. The Adaptation Server deploys the new software configuration in parallel with the existing (and still executing) configuration, restarting the mote when conditions permit. The reset command is sent from the Adaptation Server, and causes the Sun SPOT mote to reboot and begin executing the newly deployed configuration of the software. Since the data connection with the Adaptation Server will have been lost, the communications discovery protocol must execute again to re-establish the connection, and begin transmitting sensor data. Although this limited period of down-time is not ideal, it only typically lasts from approximately 2 to 3 seconds. Within a large scale wireless sensor network with many concurrently executing motes, a small period of downtime to update the functionality of a mote may be much more preferable to continuing execution with a poorly performing sensor application.

4.4 Summary

This chapter has described the implementation of the TimePredict approach, using the scenario of a dynamically-adaptable sensor application executing on a resource-constrained Java Sun SPOT mote. The primary aim of this scenario is not to demonstrate any method of dynamically-adapting software, but facilitate the analysis of a run-time timing analysis process executing on a resource-constrained device. Since the Sun SPOT sensor application described in this scenario can change its functionality at run-time, and since the current configuration of this software is determined through forecasts of its timing behaviour, the TimePredict approach can provide the optimization function within each mote to guide adaptation selection.

The implementation of the various predictive models used within TimePredict was outlined in this chapter, and an initial evaluation of their performance using a test framework was described. Lastly, the role of the timing estimates as feedback into the adaptation process was described, and an experimental scenario presented to determine the accuracy and overhead of TimePredict as a measurement-based timing analysis method for dynamically-adaptable software systems. The next chapter describes this evaluation process, and analyzes the various operational characteristics of the TimePredict approach.

Chapter 5

Evaluation

Exitus acta probat.

The result justifies the deed.

Ovid (Publius Ovidius Naso)

The use of TimePredict within a dynamically-adaptable wireless sensor mote, provides a means of evaluating the accuracy and precision of timing estimates generated within a highly restricted, yet functionally variable operating environment. As well as assessing predictive performance, this scenario offers an appraisal of the general suitability of reactive measurement-based timing analysis as a forecasting method within resource-constrained and embedded systems. Both the functional and non-functional behaviour of TimePredict must be closely examined, since a very precise or highly accurate predictive process is insufficient if it is also excessively demanding of the underlying system resources, or it fails to cope with the intrinsic variability implicit within dynamically-adaptable software. The evaluation of the TimePredict approach, presented in this chapter, analyzes its capabilities as a predictive method, its impact on the target operating environment, and its possible application as a feedback mechanism into a run-time adaptation selection process. This evaluation includes,

1. A determination of the overall predictive performance of TimePredict, executing on a dynamically-adaptable sensor mote.

2. An assessment of the impact of TimePredict on the operation of the underlying system,

and its use of the limited system resources.

3. An examination of the potential of timing estimates to be used as feedback into the adaptation selection process.

4. An evaluation of the effects of TimePredict model parameters on forecasting accuracy and precision, using simulated timing measurements.

The performance of TimePredict can be described as a function of the accuracy and precision of its estimates, its efficiency in the use of the available system resources and its impact on the normal execution-time performance of the software. Ideally, a timing analysis process would correctly forecast the worst-case and average-case timing behaviour of the software, with a high-level of precision, and without impacting normal operations or consuming any system resources. However, in practice, a very high-level of predictive accuracy is not possible without poor overall precision, since the timeliness of software can vary somewhat randomly, under the influence of many subtle factors including I/O interrupts, hardware failures, networking issues and contention for shared system resources. In common with other software tasks executing on a single processor system, TimePredict must make some demands of the limited processing and memory storage capabilities provided by the target hardware platform, in this case a Java Sun SPOT mote. The interleaving of analysis instructions on the processor, or the use of discrete blocks of memory within the system, can disturb (however slightly) the usual timing behaviour of the software being measured.

The remainder of this chapter presents an evaluation of the predictive performance of TimePredict, and its impact on a dynamically-adaptable sensor analysis function executing on a live Java Sun SPOT mote. The timing estimates generated for each of the 16 potential configurations of the sensor function are analyzed, and their overall accuracy and precision described. These timing estimates are then correlated against the range and variation of execution times within each configuration of the system. The overhead of TimePredict, both in terms of memory usage and perturbations on normal software operation is examined and a number of statistical analysis techniques applied in order to reveal the extent of the impact on the underlying system. These statistical analysis techniques are described in the next section, and are also summarized in the Glossary. The execution time of the TimePredict analysis function itself is assessed, and the effects of its ongoing execution are evaluated with respect to the power consumption and memory utilisation on the Sun SPOT mote. Additional analysis of the operation and performance

of TimePredict is described, with an evaluation of the effects of different TimePredict settings (e.g., lag length and array size) on the accuracy and precision of timing estimates, using the four simulated software timing distributions previously introduced in the Chapter 4.

Claimed Contribution	Supporting Evaluation
Operate within a resource-constrained embedded device.	Evaluation of memory consumption, power consumption and processing overhead.
Limit the impact on the system due to the timing analysis.	Assessment of the memory and processing overhead both with and without TimePredict deployed to the mote.
Accurate and precise estimates of software timeliness.	Evaluation of lag length/array sizes on accuracy and precision, and their impact on performance.
Accuracy and precision only slightly lower than what could be achieved using a detailed off-line analysis.	Comparison of off-line generated timing estimates against run-time generated equivalents within TimePredict.
Timing feedback provides actionable information to assist the software adaptation process.	Analysis of timing behaviour within the overall scenario.

Table 5.1: Summary of the expected contribution matched to particular evaluations.

The claimed contributions of TimePredict as part of a dynamically-adaptable system are outlined on the left of Table 5.1, and matched with evaluations presented later in this chapter, in order to determine whether the goals of the TimePredict approach have been demonstrably achieved. The first four contributions can be evaluated numerically to determine the veracity of the claims. The remaining contribution, on timing feedback, can be evaluated in terms of the overall performance of TimePredict within a dynamically-adaptable resource-constrained system.

5.1 Off-line execution using benchmark measurements.

A set of time series data, described in Section 4.2.3, was recorded to provide the basis for an off-line evaluation of TimePredict.

	ES Model		ES Parameters		WCET Heuristic		GEV Model	
Config.	Acc. (%)	Prec. (ms)	α	β	Acc. (%)	Prec. (ms)	Acc. (%)	Prec. (ms)
1	0.97	3.82	0.128	0.005	1.00	19.34	1.00	18.43
2	0.99	8.58	0.055	0.022	1.00	15.90	1.00	17.61
3	0.86	1.63	0.113	0.006	1.00	11.98	1.00	10.86
4	0.82	16.83	0.097	0.003	1.00	16.18	1.00	17.08
5	0.83	7.54	0.110	0.006	1.00	12.18	1.00	10.16
6	0.91	18.13	0.095	0.021	1.00	11.95	1.00	12.21
7	0.91	2.12	0.108	0.030	1.00	5.04	1.00	3.73
8	0.98	123.48	0.082	0.003	1.00	31.27	1.00	23.20
9	0.92	4.32	0.057	0.007	1.00	22.59	1.00	12.52
10	0.95	9.87	0.061	0.003	1.00	16.26	1.00	13.46
11	0.90	3.43	0.074	0.006	1.00	20.62	1.00	12.19
12	0.84	41.61	0.099	0.003	1.00	60.93	1.00	59.31
13	0.97	4.54	0.051	0.006	1.00	14.14	1.00	10.09
14	0.90	13.79	0.111	0.003	1.00	12.41	1.00	11.60
15	0.96	16.34	0.165	0.101	1.00	10.49	1.00	9.68
16	0.90	36.08	0.103	0.003	1.00	36.61	1.00	26.63
17	0.95	7.25	0.063	0.009	1.00	24.91	1.00	12.58
18	0.96	9.76	0.056	0.003	1.00	17.43	1.00	13.46
19	0.87	8.59	0.105	0.008	1.00	25.78	1.00	15.18
20	0.85	131.59	0.061	0.003	1.00	31.15	1.00	17.07

Table 5.2: Off-line analysis performed by TimePredict on benchmark measurements.

Table 5.2 describes the off-line analysis performed on a series of 20 configurations of a software benchmark suite. Over the 20 benchmark configurations, the ES model displayed an average accuracy of 91.3%, with an overall precision of 23.4 milliseconds. The smoothing parameters α

and β were 0.089 and 0.012 over the 20 configurations, obviously changing as required to match the various timing behaviours apparent within the 20 benchmark configurations. The worst-case heuristic bound encapsulated 99.899% of timing measurements over the course of 20,000 estimates, with GEV model showing a slightly superior predictive performance, encapsulating 99.964% of measurements. Similarly, the precision of the GEV model was better than that of the WCET heuristic, with an average precision of 16.35 milliseconds compared to 20.87 for the heuristic estimates.

The off-line analysis is useful in that it displays the forecasting performance of the TimePredict approach on its own, by de-coupling the analysis process from the usual operations of the system. Estimates having an average accuracy of over 90%, given the changeable nature of the underlying timing behaviour (see Figure 4.10), are sufficient for the type of applications considered within the scope of this thesis. However, the off-line analysis does not expose one of the principal threats to validity of the overall TimePredict approach, namely, the effect of any timing analysis on a resource-constrained operating environment. Embedded software typically provides a timely, reactive, and predictable response to user inputs as they occur at run-time [Lee, 2002]. The functionality is designed around the restrictions of the operating environment so that processing unanticipated tasks does not adversely affect normal operations. However, since embedded systems typically contain a single processor, with a small cache size and reduced memory storage, any timing analysis performed must be interleaved with other operations. Where this interleaving negatively effects the timeliness of the software, or other non-functional functional properties of the system (e.g., its memory usage, throughput or reliability), the benefits of accurate estimates may be nullified by the excessive impact the estimation process has on the software.

Similarly, TimePredict assumes the execution times of each configuration within a dynamically-adaptable system have a constant variance. In other words, a set of timing measurements taken immediately after a configuration begins executing are assumed to have the same statistical properties as another set of measurements from the same configuration recorded much later. Although TimePredict is evaluated against various configurations with non-uniform timing behaviours (e.g., see Configurations 6, 7 and 8 in Appendix A), an extremely variable timing process may defeat a run-time average-case or worst-case analysis approach. Although TimePredict's forecasting methods are predicated upon the assumption of past timing behaviour being indica-

tive of future trends, the effects of non-constant variance within timing measurements will affect the precision more so than the accuracy of any timing estimates. For instance, the GEV/GPD worst-case estimates, as well as the ES model, can quickly react to large variations in timing behaviour with similar increases in the WCET or ACET bounds. However, evaluations of various non-constant timing behaviours, as described later in Sections 5.3.1 and 5.3.2 show TimePredict is capable of forecasting even these difficult execution time behaviours in an accurate, precise manner.

5.2 Analysis Setup

For the experiment, a dynamically-adaptable sensor analysis function iterates over a set of sensor readings, and performs a series of complex calculations on the accumulated sensor data, before transmitting a summary of the data back to the Adaptation Manager. The execution time of this sensor analysis function will be determined largely by the amount of sensor information it records, the type of analysis performed on the data, and the overall sample size of the collected sensor readings. In addition to altering the timeliness of the software, more system's resources will be consumed processing larger datasets, i.e., the arrays used to store sensor information will be larger and take longer to iterate over. Both the execution time and system usage information is recorded concurrently with the execution of the sensor analysis function, and transmitted to the Adaptation Server to be stored in a log file for later analysis.

Since a single iteration of the sensor analysis function does not provide a good measure of the central tendency of its overall execution time, 2,000 iterations of each configuration of the software were performed, and their execution times recorded. Ideally, the timing behaviour for each software configuration should be assessed by evaluating an infinite number of timing measurements. However, for practical purposes, a more limited number of measurements must suffice instead. The greater the number of measurements recorded, the better will be its representation of the actual process being evaluated (i.e., software timeliness). For example, 3 or 4 timing measurements are insufficient to calculate the mean of a particular process with any degree of confidence. Conversely, recording millions of measurements may be unnecessary if the variation within each measurement is very small. A sample size of 2,000 measurements allows summary statistics such as the maximum, minimum and mean values to be approximated to a reasonably high level of confidence, i.e., the difference between the mean of the recorded timing measure-

ments and the actual mean timing behaviour of a particular software configuration is likely to be be small. Typically, as the sample size increases, the mean of the sample converges to the mean of a conjectured dataset containing an infinite number of measurements.

The analysis of the combined 32,000 timing measurements is presented in the Section 5.3.1, using one 2,000 measurement dataset for each of the 16 configurations of the system. As well as exposing the timeliness of the software, these timing measurements are used as inputs during run-time into the TimePredict estimation process. In turn, the timing estimates generated by TimePredict are likewise logged by the Adaptation Server, and are compared against the next timing measurement to assess their predictive accuracy, as well as their overall precision. The impact of TimePredict on the underlying system is evaluated off-line in Section 5.3.3, by comparing the differences in execution times and resource usage, between two similar configurations of the software, one deployed with the TimePredict functionality and one deployed without. The results of this statistical analysis of the TimePredict overhead are presented in Section 5.3.3, and the various statistical tests used to make this determination are presented in the next section.

5.2.1 Statistical Analysis Tools

The output of the TimePredict testing process consists of a series of numeric measurements, grouped into several different sets, and analyzed to determine if statistically significant differences exist between distinct groups of measurements. For example, it can be observed that the execution time of each configuration of the sensor analysis function is different, however a statistical evaluation must be performed in order to determine the extent of any differences that may exist between the accuracy and precision of the various predictive models, as well as highlight the overall impact of TimePredict on the underlying system.

A number of metrics are recorded by the dynamically-adaptable sensor application during execution and transmitted to the Adaptation Server for logging (due to memory restrictions on the mote itself). The metrics are recorded in a separate log file for each configuration of the software, i.e., whenever an adaptation occurs a new log file is created for the next configuration of the sensor analysis function. The metrics saved in these log files include the execution time of the sensor analysis function, the bounded worst-case and average-case timing estimates for each of the predictive models in TimePredict, the remaining charge in the mote battery in milliampere-hours, and the ratio of free memory to total memory available on the mote. The data within

these log files forms the basis for the statistical evaluations described in this chapter, with the exception of the simulated timing measurements within Section 5.4.2.

The experimental scenario forces an adaptation of the system after 2,000 iterations of the sensor analysis function, by setting a timing threshold value within the Adaptation Manager on the mote and thereby initiating the deployment of a new configuration of the software. This artificial adaptation trigger was required to guarantee a sufficient number of timing measurements could be gathered, and at the same time ensure that an adaptation would eventually take place. The somewhat random variation within the execution time of the software makes it difficult to predict the interval between adaptations using a purely time-based adaptation mechanism. However, once the software has completed 2,000 iterations and the log data has been collected, a series of statistical tests can then be applied. The aim of this off-line statistical analysis and evaluation process, and indeed the aim of this chapter, is to establish the obvious systematic differences that may exist between the various predictive models used within TimePredict, as well as the overall impact of the timing analysis process on the system. The comparisons used to explore the performance of TimePredict, as well as the characteristics of the dynamically-adaptable sensor application, include;

- An analysis of the differences in the range of execution times and their central tendency for each configuration of the software.
- An evaluation of the performance of the various predictive models within TimePredict across each configuration of the software.
- A determination of the relative performance of each predictive model against the number of timing measurements available within the system.
- A determination of the impact of TimePredict on the operation of the software, and the performance of the underlying system.

A number of software packages were used to the perform the basic statistical analysis, as well as the more complex hypothesis testing. A spreadsheet was used to calculate basic descriptive statistics, e.g., the mean, standard deviation, and quartile values, with more complex statistical tests carried out using two dedicated statistical software packages, namely, MiniTab 16 [Minitab, 2010] and the R statistics package [Chambers, 2008]. These statistical packages were mostly

confined to finding correlations between datasets as well as producing a series of hypothesis tests, to determine whether a statistically significant difference exists between two groups of values. The confidence level for the various statistical tests was set to 95%, and the p-value of each test provided where appropriate. The p-value is the probability of getting the same, or a more extreme result, assuming that the null hypothesis is true. For the purposes of timing analysis, the null hypothesis is that no difference exists between two sets of timing measurements. When there is insufficient evidence to reject the null hypothesis, we must instead accept it, and conclude there is no discernable data to show a statistically significant difference in the two sets of measurements [Upton and Cook, 2004].

5.2.1.1 Independent Two-Sample T-Test

The two-sample independent t-test is a statistical test to find whether the means of two sets of normally-distributed values are statistically different. In addition to assuming normality within both sets of measurements, each value is presupposed to have the same variance and to be independent, i.e., the result of a measurement in one set does not depend on any value in the second set. The power of the t-test is its evaluation of systematic variation between two groups of measurements, while trying to reduce the effects of random noise due to variation within each group. However, the result of the t-test will be valid only if its assumptions about the analyzed data hold true.

5.2.1.2 Non-Parametric Tests

The Mann-Whitney test is a two-sample non-parametric statistical test to determine whether a statistically-significant difference exists between two sets of measurements [Upton and Cook, 2004]. Non-parametric tests, such as the Mann-Whitney or Mann-Whitney U test, are useful in that they make no assumptions about the underlying data such as normality, or having similar means or variances. The null hypothesis for the Mann-Whitney test is that both samples come from the same population to a 95% level of confidence, i.e., the processes that generated both datasets were identical. By rejecting the null hypothesis, the test effectively states that there is a noticeable and significant difference between the measurements within the two datasets.

The Mann-Whitney test is provided in addition to the two distribution-based t-tests, as an additional assurance in the correctness of the t-test results. Where both the Mann-Whitney non-

parametric test, and the model-based t-tests provide a similar evaluation of the underlying data, the fundamental confidence in this evaluation is enhanced. Within the analysis of the performance of TimePredict, the Mann-Whitney test is used as an adjunct to the t-tests, especially within the determination of the extent of the impact of the TimePredict approach on the underlying system.

5.2.1.3 Correlation Tests

Correlation is a measure of the linear association that may exist between quantitative variables, such that a change in one variable is matched by a reciprocal change in the other. Correlations can be positive or negative, i.e., an upward change in one variable being matched by either an upwards or downwards change in the other. This can be used to indicate the departure from independence of specific facets of a particular process. Correlation testing is performed by calculating the Pearson correlation co-efficient for two sets of values [Upton and Cook, 2004], using either a spreadsheet or a statistical software package. The Pearson correlation coefficient can range between -1.0 and 1.0, with values typically within the range -0.3 to 0.3 indicating little or no relationship between the variables.

5.3 TimePredict Evaluation

The evaluation of the performance of TimePredict begins with an evaluation of the intrinsic timeliness of the software executing on the sensor mote. Timing measurements taken of the dynamically-adaptable sensor analysis function form the basis of the timing estimates produced by TimePredict. The various configurations of the sensor software are used to examine the predictive performance under different timing behaviours, e.g., highly variable timing behaviours with a large range may be more difficult to accurately predict. The next section (Section 5.3.1) evaluates the observed timing behaviour of the 16 configurations of the sensor software function, each deployed with TimePredict, and executing within a live operating environment on a resource-constrained device.

Once the expected timing behaviour of the software has been established, Section 5.3.2 analyzes the performance of TimePredict, and describes the overall accuracy and precision of its timing forecasts for each of the configurations of the sensor software. Lastly, Section 5.3.3 evaluates the impact of TimePredict on the underlying system, and describes the various statistical

analyses performed to assess the overhead in terms of operational interference, memory overhead and power consumption.

5.3.1 Software Timeliness

Table 5.3 presents a summary of the execution times of each of the 16 configurations of the sensor analysis application, the configurations having been described previously in Table 4.3.

Config. No.	Mean (ms)	Min (ms)	Max (ms)	Range (ms)	Q_1	Q_3	IQR
1	49.72	47	51	4	50	50	0
2	58.58	49	65	16	59	59	0
3	72.92	51	85	34	74	74	0
4	74.57	51	85	34	75	76	1
5	58.72	47	68	21	61	62	1
6	138.37	49	182	133	167	170	3
7	139.14	49	181	132	167	171	4
8	257.06	50	371	321	323	326	3
9	67.93	65	72	7	68	68	0
10	85.58	67	96	29	86	86	0
11	112.78	69	124	55	114	114	0
12	120.37	71	151	80	121	121	0
13	77.82	58	94	36	84	84	0
14	238.25	60	317	257	297	303	6
15	461.14	62	626	564	592	598	6
16	513.31	70	695	625	659	664	5

Table 5.3: Evaluation of the execution time of the sensor analysis function.

This table, in contrast to the previous summary table (Table 4.2) showing the benchmark data, instead describes the average, minimum, maximum and inter-quartile range of the sensor application timing performance, as recorded at run-time. These execution times themselves are further illustrated within Appendix A. The graphs in this Appendix show the variation within the timing data of the 16 software configurations summarized in Table 5.3. Although the software continues

execution immediately following an adaptation, for the purposes of this evaluation each configuration of the software is considered separately. The mean execution time shows the software timeliness generally increasing as the sensor analysis function becomes more computationally demanding, i.e., from Configuration 1 to Configuration 16.

The minimum and maximum execution time measurements illustrate the range of potential timing behaviours. The first and third quartile values are listed as Q_1 and Q_3 respectively, and represent the timing measurements that bound 25% and 75% of the measurements respectively. The inter-quartile range (IQR) is the difference between these quartile values, and is used to illustrate the level of clustering (or not) within the middle 50% of the recorded timing measurements, with a value of 0.0 representing a highly clustered distribution. Since each configuration of the software begins execution with no previous sensor measurements available to it, and hence without any populated data-structures to iterate through, the minimum execution time of each configuration is roughly similar. However, the minimum execution times presented within Table 5.3 are very highly correlated with the number of sensors used by the analysis function, suggesting that extra sensor functionality adds an immediate additional overhead on the execution time of a particular configuration of the software. Similarly, the sample size of each software configuration shows a strong positive correlation with the mean, maximum, range and inter-quartile range of the observed timing behaviour for each configuration, i.e., the greater the sample size, the higher these values tend to be within Table 5.3.

	Mean (ms)	Min (ms)	Max (ms)	Range	Q_1	Q_3	IQR
Sensors Used	0.322	0.923	0.303	0.265	0.375	0.302	0.233
Analysis Complexity	0.499	0.259	0.498	0.492	0.526	0.484	0.440

Table 5.4: Correlations of configuration setup with execution time analysis.

Table 5.4 describes the correlation between the parameters used to differentiate the various software configurations, and the observed impact these parameters appear to have on the subsequent timing behaviour, i.e., the correlation coefficients presented in the table hint at the apparent causes of timing variation between the different configurations of the software. The correlation coefficients must lie within the range 1.0 to -1.0, with a strong correlation defined as any value in excess of 0.7 or less than -0.7 and a weak correlation indicated by values between -0.3 and +0.3. While the values in Table 5.4 have no direct impact on the TimePredict estimates,

they may be used to link increases or decreases in predictive performance with an underlying functional cause, e.g., where more sensors are used within a particular configuration we can expect its minimum execution time value to be greater. Identifying the expected timeliness of the each configuration of the software is not important in itself, in so far as it demonstrates that each configuration of the software has a unique timing behaviour, with a different range of potential timing values and contrasting levels of clustering within these values. For example, the sample size used in the sensor analysis, although restricted to a tuple value (100/1000) and therefore inappropriate to use as a correlation coefficient, can be compared to show its effect on the mean timing for the various configurations. For example, Configurations 12 and 16 differ only in the number of sensor measurements they calculate (see Table 4.3 from the previous chapter), yet the difference in their mean timing values is 392.94ms. In contrast, a similar comparison between Configurations 4 and 8 yields a difference of 182.49ms. These subtle differences in timing behaviour are used to test the predictive accuracy and precision of TimePredict, using only run-time measurements as the basis for the analysis process, i.e., not having any prior knowledge of the expected timeliness of the software.

The timing differences between the various configurations are somewhat opaque when presented numerically, but are more readily accessible when illustrated as a box-plot, as shown in Figure 5.1. This box-plot describes the data in Table 5.3, with the minimum/maximum values shown as bounded lines, and the inter-quartile range represented as boxes. The mean value is shown as a black horizontal line, with the median value (not shown) being the mid-point of each box. The most apparent characteristic within Figure 5.1, is the extended execution time range of configurations with a large sample size, i.e., configurations 6 to 8, and 14 to 16. The amount of variation within these configurations also appears to be greater, as is illustrated with the larger maximum/minimum range of the timing measurements, as well as a higher inter-quartile range.

The differences in both range and clustering are likewise apparent, with configurations 1 to 5, and 9 to 13 having a very compact range of execution times more clustered about the mean. This type of stable timing behaviour should prove relatively straight-forward to forecast at run-time, since the variation inherent within the timing measurements is somewhat more constrained. The next section describes the performance of TimePredict in estimating the execution times of these software configurations.

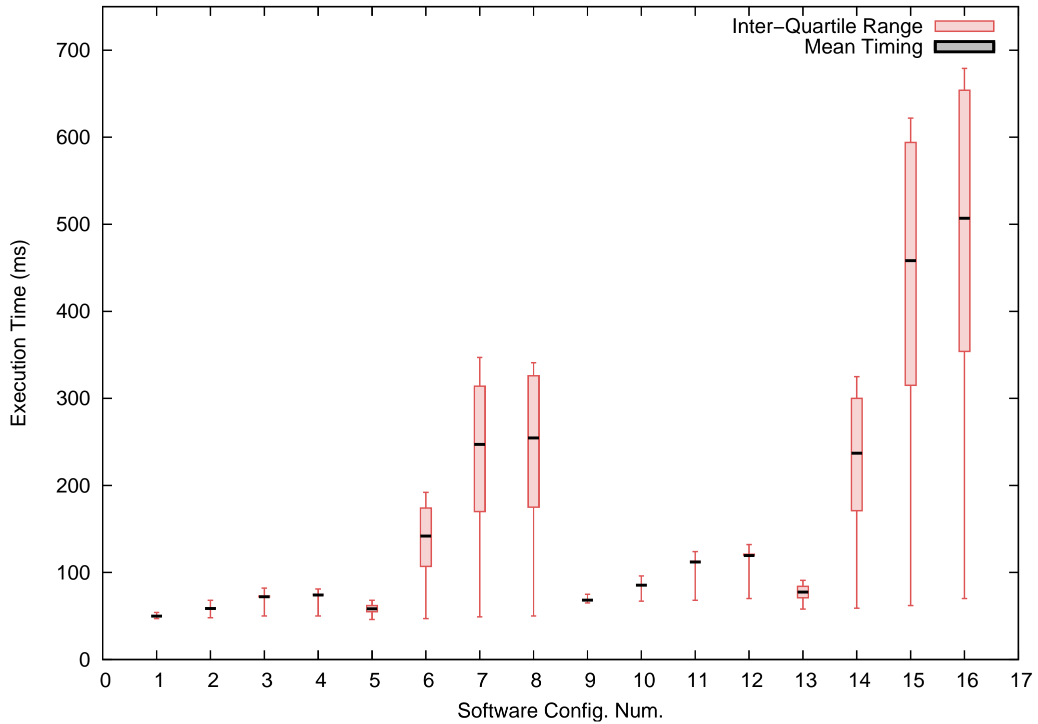


Fig. 5.1: Box-plot showing the maximum and inter-quartile range of execution times.

5.3.2 TimePredict Performance

The percentage accuracy of TimePredict can be assessed by determining whether the bounded ACET and WCET timing estimates encapsulate the execution time of the software. Similarly, the precision of the estimates can be determined by calculating the difference in milliseconds between the estimate range and the actual timing measurement. The timing bounds for both the ACET and WCET estimates move whenever there is variation within the underlying timing measurement data. As illustrated within Appendix A, the predictive models become more pessimistic whenever a previously established ACET/WCET timing bound is violated, leading to periodic increases or decreases in accuracy and precision during run-time. For the purposes of this evaluation, the overall accuracy and precision of TimePredict is examined, with the performance of each predictive model assessed over the course of 2,000 measurements of 16 separate software configurations.

Table 5.5 shows the predictive accuracy (Acc.) of TimePredict in percentage terms, across

	ES Model		ES Parameters		WCET Heuristic		GEV Model	
Config.	Acc. (%)	Prec. (ms)	α	β	Acc. (%)	Prec. (ms)	Acc. (%)	Prec. (ms)
1	93.83	4.29	0.925	0.003	99.73	2.91	100.00	11.58
2	93.38	4.28	0.811	0.001	99.81	11.06	99.91	8.17
3	94.08	5.28	0.939	0.003	98.54	13.45	99.90	10.33
4	93.09	5.25	0.861	0.002	99.01	15.18	99.94	11.42
5	93.69	4.58	0.928	0.002	99.65	10.15	99.85	7.71
6	92.84	5.84	0.805	0.002	96.42	15.26	99.92	14.28
7	92.94	6.08	0.917	0.002	99.48	26.07	99.94	20.13
8	92.46	29.79	0.804	0.002	98.88	86.23	99.83	55.79
9	99.63	2.60	0.910	0.001	99.86	14.98	100.00	11.65
10	99.49	3.03	0.822	0.003	97.73	16.77	99.86	12.04
11	99.66	3.03	0.968	0.001	99.36	14.35	99.80	9.77
12	96.00	12.96	0.943	0.003	99.37	56.20	99.80	32.37
13	99.01	3.66	0.971	0.001	99.63	17.12	99.88	10.47
14	94.61	9.06	0.947	0.003	97.41	27.83	99.65	13.91
15	93.03	14.76	0.883	0.001	99.07	55.61	99.46	18.76
16	91.71	16.80	0.713	0.003	98.55	54.98	98.85	22.38

Table 5.5: TimePredict forecasting accuracy and precision.

the 16 configurations of the software, as well as its precision (Prec.) in milliseconds. Although the accuracy of the various predictive models is calculated within TimePredict during run-time to enable ACET/WCET estimate selection, for the purposes of this evaluation an off-line evaluation is performed on each predictive model using log files (flat text files) containing the execution time and timing estimate data as tab-delimited columns. The bounded timing estimates provided by TimePredict are evaluated against the next timing measurement to assess accuracy, with the WCET estimates providing a single upper bound, and the ACET estimates both an upper and lower bound. The precision of the WCET estimates is calculated as the absolute millisecond difference between the measurement and the WCET bound, whereas the ACET precision is the difference between the upper and the lower bound of the estimate. Although TimePredict

maintains an updated run-time record of its predictive performance, the evaluation of both the accuracy and precision of its estimates was performed off-line, to enable a more detailed statistical analysis.

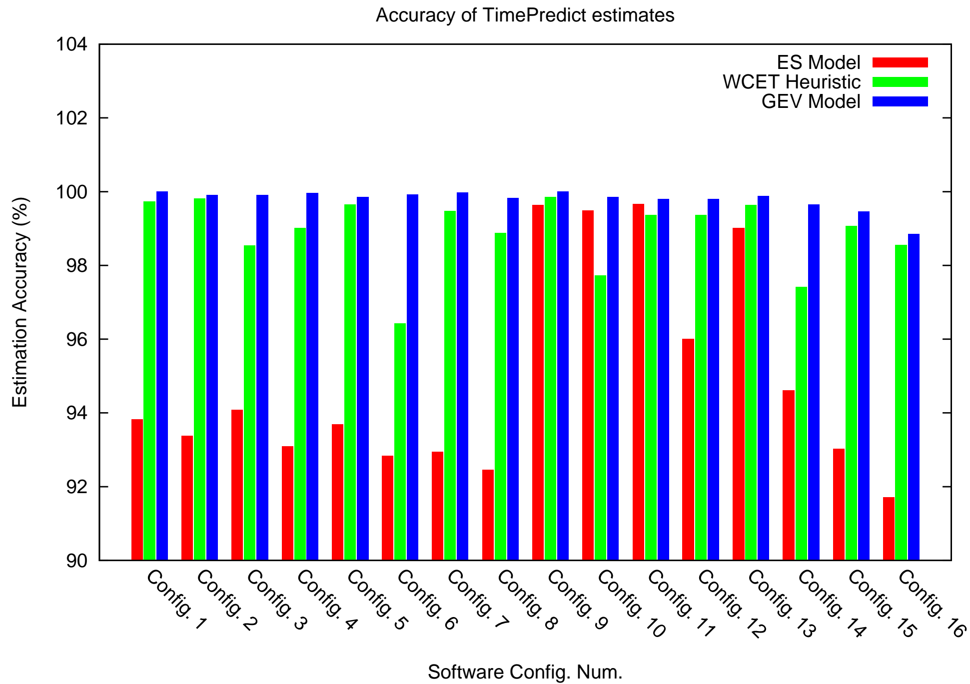


Fig. 5.2: Histogram of the estimation accuracy of TimePredict.

Figure 5.2 illustrates the accuracy of the four predictive models used by TimePredict, applied within each of the 16 configurations of the sensor software. The worst-case models tend to have a slightly higher accuracy than their average-case counterparts, due to an increased inherent pessimism, as well as their reliance on a relatively larger set of timing measurement data to produce each WCET estimate, i.e., whereas the average-case models only access the previous 50 timing measurements, the GEV model records the frequency distribution for each measurement generated within the system. The average-case predictive method, namely the ES model, has an expected accuracy of 95%, with the worst-case models expected to encapsulate 99% of the software timing measurements. The ES model performs poorly, with only 5 of the 16 configurations meeting this expected accuracy. However, the overall performance is never worse than 91.7% over the course of 2,000 measurements. The strength of the ES model however lies in its

ability to provide a timing estimate with only a very limited amount of available timing data, i.e., immediately following an adaptation to the system. Similar to the average-case estimates, the model-based worst-case timing estimates prove more accurate than their heuristic-derived counterparts, but require more timing measurements to generate a worst-case timing estimate. The GEV method out-performs the WCET heuristic, but only by an average of 0.9% over 32,000 estimates. While the GEV model meets its 99% target accuracy in 15 of the 16 configurations, it fails to meet this expected accuracy in configuration 16 by a mere 0.15%, or approximately 1 estimate in every 667, a discrepancy within the range that could be expected from random variation. As with the ACET models, the lower accuracy WCET heuristic is only intended to provide an interim WCET estimate, until sufficient measurements have been collected to facilitate GEV-based estimates. However, on its own the WCET heuristic provides an average accuracy of 98.9% over the 16 configurations of the software, while the more accurate GEV model provides an average accuracy of 99.8%.

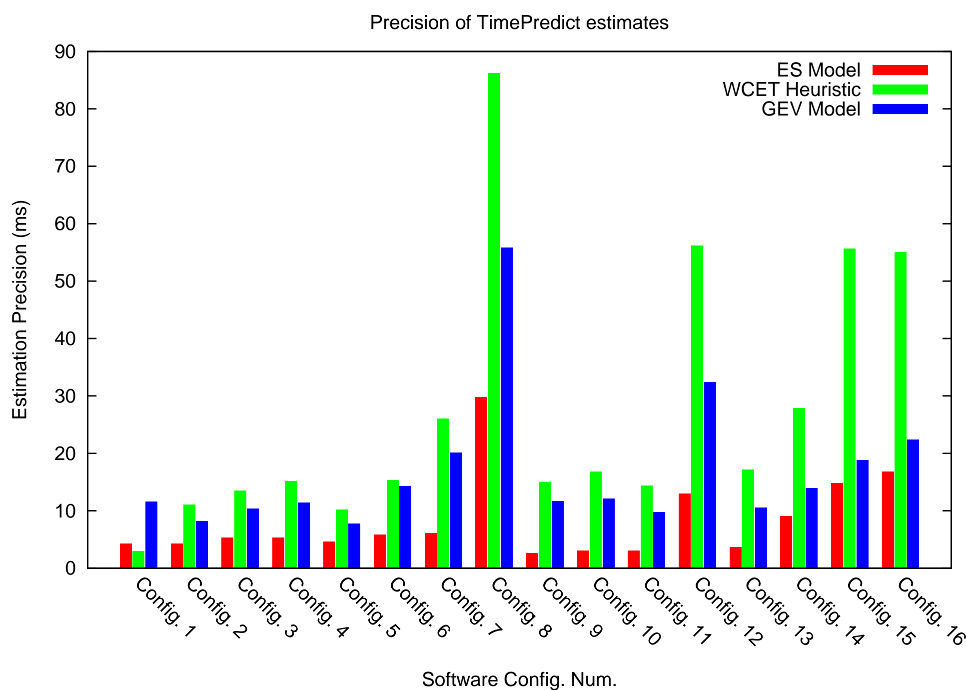


Fig. 5.3: Histogram of the estimation precision of TimePredict.

Figure 5.3 presents the precision of each of the four predictive models for all 16 configurations

of the dynamically-adaptable sensor function. Timing estimates produced for configurations 8, 12, 15 and 16 exhibit the poorest overall precision, with the worst-case forecasting methods typically providing a more pessimistic (and less precise) overall estimate. Again, the WCET heuristic method compares unfavourably to the model-based GEV forecasting method within the same configuration of the software, due primarily to its use of the observed maximum execution time when calculating the WCET bounds. Although the vertical axis of Figure 5.3 shows the average precision in milliseconds, the correlation between this precision and the expected timing range of each configuration must also be considered.

	ES Model		WCET Heuristic		GEV Model	
Correlation	Acc.	Prec.	Acc.	Prec.	Acc.	Prec.
Range	-0.486	0.702	-0.240	0.724	-0.868	0.442
IQR	-0.446	0.531	-0.240	0.560	-0.901	0.229

Table 5.6: Correlation between accuracy/precision and the overall range and IQR.

Table 5.6 presents the correlations between both the accuracy and precision of the forecasting methods, and the range and inter-quartile range of the various software configurations. This table summarizes the effects of timing variation on the overall predictive performance of TimePredict, by relating the columns in Table 5.5 with the Range and IQR values in Table 5.3. This correlation is important, since it can be used establish how timing variation affects the accuracy and precision of timing estimates. For example, the range of potential timing measurements appears to be an important determinant of worst-case accuracy and precision within the GEV model. As the range of potential timing measurements increases, the probability of the worst-case execution time begin exposed through repeated measurements decreases, leading to uncertainty about the extent of the worst-case timing behaviour for the software. Since the GEV model relies on a relatively large number of timing measurements to generate an estimate, a relatively larger range effectively dilutes the information that can be obtained from a given number of measurements. The GEV model shows a strong negative correlation between these ranges and its predictive accuracy. In effect, as the range of potential execution times increase, the accuracy of the model decreases. This may be explained by the worst-case timing behaviour being encountered with less regularity over the course of several thousand timing measurements, if the potential range of

timing measurements is very large and the variability between measurements is also high. Since the GEV model relies on timing measurements to generate estimates, a relatively larger range within these measurements will require more observations to generate estimates to the same level of predictive accuracy.

5.3.3 System Impact

The impact of TimePredict on the underlying system can be established through measuring the performance of two complementary configurations of the software, under similar conditions, and on the same hardware platform. The two software configurations differ only in being deployed with or without the TimePredict functionality as part of the overall software package executing on the mote. By statistically evaluating the difference in the timeliness and memory usage of both configurations, the effects of the run-time timing analysis on the underlying system may be discerned, within a stated level of confidence. This statistical estimate for the difference in execution times as well as memory overheads provides a measure of the overall suitability of TimePredict for operation within resource-constrained and embedded devices. The next section introduces the execution time overhead of the TimePredict function itself, with Sections 5.3.3.2 and 5.3.3.3 describing its estimated memory usage and power consumption respectively.

5.3.3.1 Timing Overhead

TimePredict must consume system resources in order to generate estimates, since the actions of recording timing data, fitting statistical distributions, and performing goodness-of-fit tests all take up processor cycles and memory. On a single-processor system, this means either the static scheduling or dynamic weaving of timing analysis instructions with 'normal' processing on the CPU. For the experimental evaluation of TimePredict, the timing analysis was executed immediately after the execution of the sensor analysis function, in order to minimize the context switching between the two tasks within the system. However, other factors such as resource allocation and de-allocation, garbage collection and changes in the execution history, may induce sympathetic delays within the timing analysis functionality.

Figure 5.4 shows the execution time of the TimePredict function itself, over a period of $2,000$ iterations. Each iteration represents a single call to the TimePredict function to generate forecasts using all four predictive models available within the system. After each timing estimate

is produced, updated measurement data is added to TimePredict, and the function called again. The execution time of the TimePredict function rises sharply as more data is added to the ACET and WCET arrays and then largely evens off once these arrays reach their full capacity, with occasional periods of increased execution times, as well as seemingly periodic timing spikes. The origin of these timing spikes is unknown, but may be an artifact of the Sun SPOT JVM performing garbage collection while the timing analysis is taking place, or could be the result of interference from a system-level periodic task on the mote. An analysis of the correlation between the execution time and the available memory provided a Pearson correlation coefficient of 0.210, indicating no relationship exists between the spikes in the execution time of TimePredict and the memory usage on the mote itself.

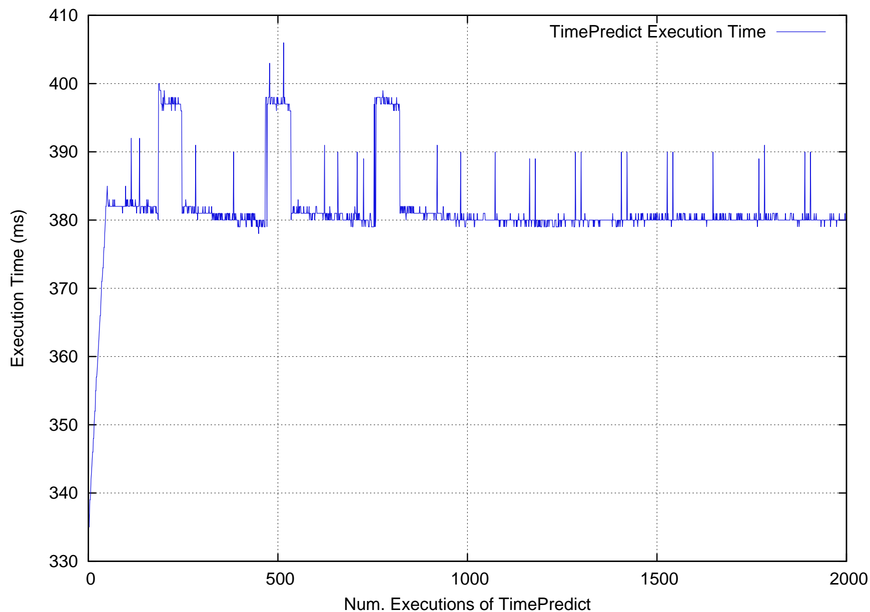


Fig. 5.4: Execution time of the TimePredict functionality on the Sun SPOT mote.

The average execution time for TimePredict, running on a Sun SPOT mote, with all predictive models enabled, is 381ms. The maximum recorded execution time was 406ms over 2,000 iterations of the software, with the minimum value recorded being 335ms. Although multi-threading is supported on the Sun SPOT motes, the experiment performed the sensor analysis and timing analysis sequentially within a single thread. Where both the timing analysis and the 'normal' software tasks are separated into different threads, TimePredict can produce an

updated timing estimate in approximately 400ms. Although the experiment had TimePredict produce a constantly updated timing estimate, the more usual trigger for the timing analysis would be to update the ACET/WCET estimates only when a software timing measurement exceeds a set threshold. This threshold-triggered analysis is supported within TimePredict in order to minimize the impact of the timing analysis process, by updating the estimates only when the timing behaviour of the software has been deemed to have changed significantly.

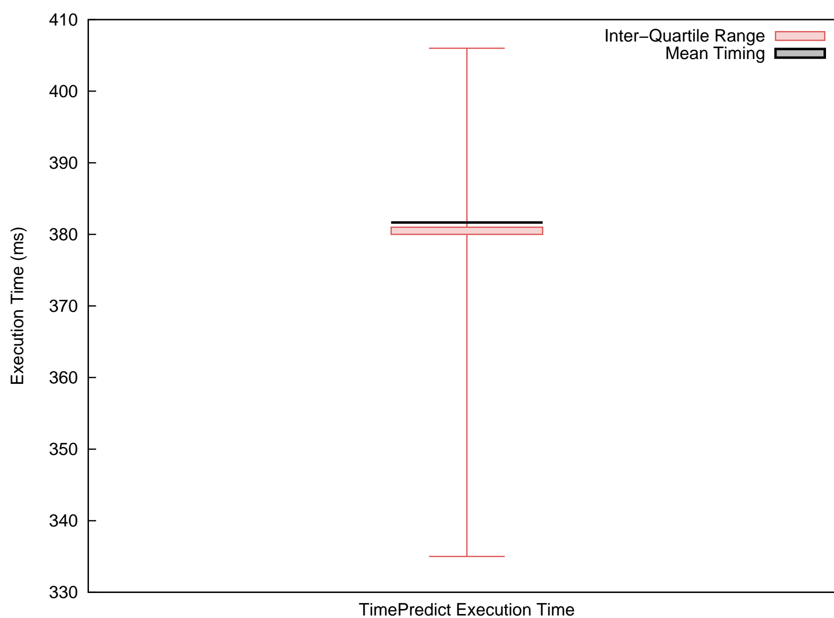


Fig. 5.5: Boxplot showing the execution time performance of TimePredict.

Figure 5.5 describes the range, and inter-quartile range of the execution times of the TimePredict framework. The minimum and maximum values (335ms and 406ms respectively) are indicated in Figure 5.5 by the thin vertical lines, whereas the inter-quartile range (from 380ms to 381ms) describes the bounds within which the middle 50% of the execution times occur. The small inter-quartile range indicates that the variation within the execution time of the TimePredict function is small, and averages at approximately 381ms. Although the sensor analysis function and the TimePredict timing analysis are scheduled to execute in such a way as to reduce any disturbance to the normal timeliness of the sensor analysis function, the deployment of TimePredict consumes system memory and processor cycles, and may result in unintentional perturbations to the normal execution time of the sensor analysis function. Table 5.7 presents

some basic descriptive statistics showing the execution time of the sensor analysis function, both with and without TimePredict deployed.

Sensor Function Exec. Time	N	Mean (ms)	SE Mean	StDev
With TimePredict	2000	513.463	4.373	195.662
Without TimePredict	2000	513.376	4.360	195.069

Table 5.7: Sensor analysis timeliness with and without TimePredict.

Table 5.7 shows that the mean, standard error of the mean, and the standard deviation of both configurations are very similar. Since the difference may be too small to detect using summary statistics, a number of statistical hypothesis tests can be carried out to establish whether there exists a statistically significant difference in the means of the two sets of execution time measurements. As introduced in Section 5.2.1, these tests consist of an independent t-test, as well as a non-parametric Mann-Whitney test. Both of these statistical difference tests starts with the assumption (null hypothesis) that no difference exists between the two datasets, and then sets out to prove or disprove this assumption, and thereby accept or reject the null hypothesis.

Table 5.8 presents results of the two statistical tests performed on the timing measurements of both software configurations, showing the likely difference in the mean execution time of the sensor analysis function due to TimePredict. These tests calculate a 95% confidence interval for the difference between the two sets of measurements, as well as the test-statistic (t-value) and the significance of the test (p-value). A confidence interval describes a bounded range of values that encapsulates the mean difference between the two sets of measurements, with a 95% probability. The t-value is the test statistic used to determine whether to accept or reject the null hypothesis, where the test statistic must exceed a specified critical value to indicate a difference in the two sets of measurements. For a t-test at a 95% level of confidence this critical value is 1.96 [Lindley and Scott, 1995]. Since the test statistic of both t-tests is less than this critical value, we have insufficient evidence to reject the null hypothesis, and can find no statistically significant difference exists in the timeliness of the two configurations of the software at this level of confidence. The p-value indicates the probability (0.0 to 1.0) of obtaining a similar test statistic if the null hypothesis is true. Both p-values are very high, stating that the probability of a similar test statistic occurring randomly would be approximately 99% and 50% respectively. Typically the

null hypothesis is only rejected if the p-value is less than the significance level of the test, i.e., 0.05. In this test, the p-values of 0.98 and 0.47 are far in excess of this value, thereby reinforcing the determination that null hypothesis must be accepted - that there is no difference in software timeliness due to the deployment of TimePredict to the system.

Statistical Test	N	95% Conf. Interval	T-value	p-value
Mann-Whitney	2000	(0.00,1.00)	N/a	N/a
Independent T-test	2000	(-12.019365, 12.193191)	0.010	0.989

Table 5.8: Statistical tests for any timing interference due to TimePredict.

In addition to the t-value and p-value for the t-test indicating no difference, the 95% confidence interval for the test incorporates the zero value, giving a further indication that no detectable difference exists. In summary, there is insufficient statistical evidence to suggest a difference in the execution times of the sensor analysis function when TimePredict is deployed to the system.

5.3.3.2 Memory Consumption

In addition to measuring the execution time, the unfolding ratio of free memory to total memory was recorded for both configurations of the sensor analysis software. Similarly, these two sets of measurements provide memory usage information that can be analyzed statistically to determine whether any significant differences can be detected when TimePredict is deployed to the system. As with the execution time analysis in the previous section, the evaluation of the possible memory overhead incurred by TimePredict is performed by first analysing the basic descriptive statistics, and then applying hypothesis tests to the underlying data.

Available Memory	N	Mean (ms)	SE Mean	StDev
With TimePredict	2000	0.62511	0.00162	0.07239
Without TimePredict	2000	0.64446	0.00102	0.04579

Table 5.9: Memory usage with and without TimePredict.

Table 5.9 presents a summary of the ratio of free memory to total memory within both configurations of the system over the course of 2,000 measurements. Again, one configuration of the software was deployed with TimePredict and actively calculated timing estimates during execution, whereas the other configuration contained just the sensor analysis functionality. The memory usage ranges between 0.0 and 1.0, with 0.0 indicating a system with no available free memory. The mean values for both configurations in Table 5.9 are somewhat similar, giving an overall memory consumption of about 35% to 40%. Similarly, the standard errors for the means (SE Mean) are largely the same, however a slight difference may be perceived in the standard deviations of the two sets of measurements. This may suggest that there is increased volatility in memory usage with TimePredict deployed to the Sun SPOT mote.

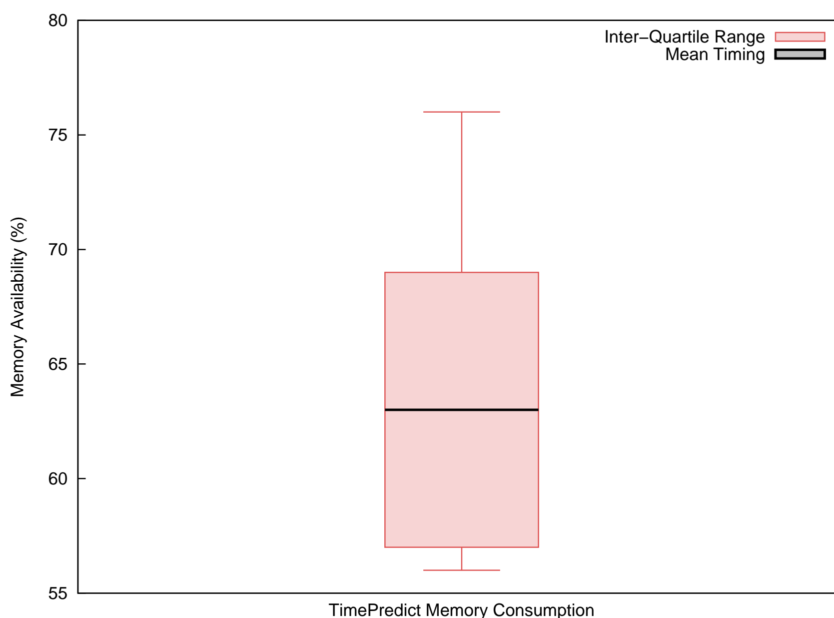


Fig. 5.6: Boxplot showing the memory consumption of TimePredict.

Figure 5.6 illustrates the boxplot for the memory consumption of the TimePredict framework. The range and inter-quartile range (IQR) are shown, and present more variation in the use of system memory than the previous analysis of TimePredict execution times (e.g., Figure 5.5). The greater inter-quartile range indicates that the extent of TimePredict's use of the available system memory can vary considerably over time, with the IQR bounds set between 57% and 69%. However, this memory usage cannot be attributed to TimePredict alone, but include

the memory usage for the underlying sensor analysis function also. The difference in memory consumption ascribed solely to TimePredict is assessed in Table 5.10. In order to determine the extent of memory usage that can be ascribed to TimePredict, statistical tests were performed to estimate the probable difference between the means of the two sets of measurements. Again, a 95% confidence interval for the difference in both datasets was generated using the same statistical hypothesis tests. The results of these tests, and their associated 95% confidence intervals, and are presented in Table 5.10.

Statistical Test	N	95% Conf. Interval	T-value	p-value
Mann-Whitney	2000	(0.05000, 0.04000)	N/a	N/a
2-Sample T-test	2000	(0.023103, 0.015598)	10.11	0.000

Table 5.10: 95% confidence intervals for memory overhead of TimePredict.

On initial examination, it appears a discernible difference does exist, since the range of each confidence interval is wholly positive, i.e., the range of the confidence interval does not encompass negative values or zero. In addition, the independent t-test provide a test statistic far beyond the critical value required to reject the null hypothesis, the null hypothesis being the assumption that no difference exists. Similarly, the low p-values provides a further indication that the probability of these test statistics being arrived at through random chance is very small. However, a normality test on the two sets of measurements suggests neither dataset is normally distributed, thus one of the assumptions underlying the t-test may be invalid. This accords slightly more significance to the result of the Mann-Whitney test, which makes no such assumptions but also suggests a statistically significant difference exists between the memory usage of software deployed with TimePredict, and configurations deployed without. Using the confidence intervals, it can be surmised that the measurements show a difference of between 1.5% to 5% in the usage of the available system memory (4593 kB), or an additional memory overhead for TimePredict somewhere within the range 69 kB to 230 kB. Within a resource-limited system, with approximately 65% of its available memory unclaimed during execution, an additional overhead of 1.5% to 5% to support the operation of TimePredict appears to be sustainable.

5.3.3.3 Power Consumption

The battery within the Sun SPOT mote nominally provides 720 milliampere-hours (mAhr) when fully charged, meaning that the mote power source can theoretically supply a current of 720 milliamps for a period of one hour before becoming fully discharged. In practice, the power requirements for typical Sun SPOT operations are much less than this, with the application developer trying to limit the power consumed by the radio and LEDs, in order to prolong the operational lifetime of the device on a single charge. The Sun SPOT has a stated current draw of approximately 104 milliamps, when actively calculating with the radio and sensors enabled, and a minimum current draw of 24 milliamps in shallow sleep, and 33 microamps in deep sleep mode, according to the Sun SPOT hardware release notes [Sun, 2009]. This provides an assumed operational lifetime, under a single battery charge, of between 3 hours and 900 days, depending on the level of activity within the device.

During the course of the experiment using the dynamically-adaptable sensor application, one of the parameters logged within the Adaptation Server and broadcast by the Sun SPOT mote was the remaining batter power in milliampere-hours (mAhr). Unlike the theoretical maximum charge of 720 mAhr, the repeated charging and discharging of the tested Sun SPOT mote had degraded the performance of the battery, so that it yielded a maximum battery charge of 713 mAhr. Although the LEDs on the mote were switched off to conserve power, the drain on the battery was considerable during testing, due to the constant radio communication, and the complex calculations carried out on the mote. Figure 5.7 illustrates the power consumed executing a single configuration of the software, over a period of 400 minutes (approximately 7 hours). The power consumed both with and without TimePredict deployed on the mote was the same, as the software continuously occupies the processor with calculations, and the radio with transmissions in either case. The level of battery power used by each of the 16 configurations of the software was also the same, with a Pearson correlation co-efficient of 0.9999 between the power consumption of each configuration of the system (a coefficient of 1.0 would signifying completely identical datasets).

As shown in Figure 5.7, the average rate of power consumption while executing a configuration of the software is 1.74 mAmps per minute, or 104.12 mAmps per hour (matching the current draw stated by the manufacturer). The power usage was calculated using two configurations of the software (with/without TimePredict), running on the same Sun SPOT mote as used for the

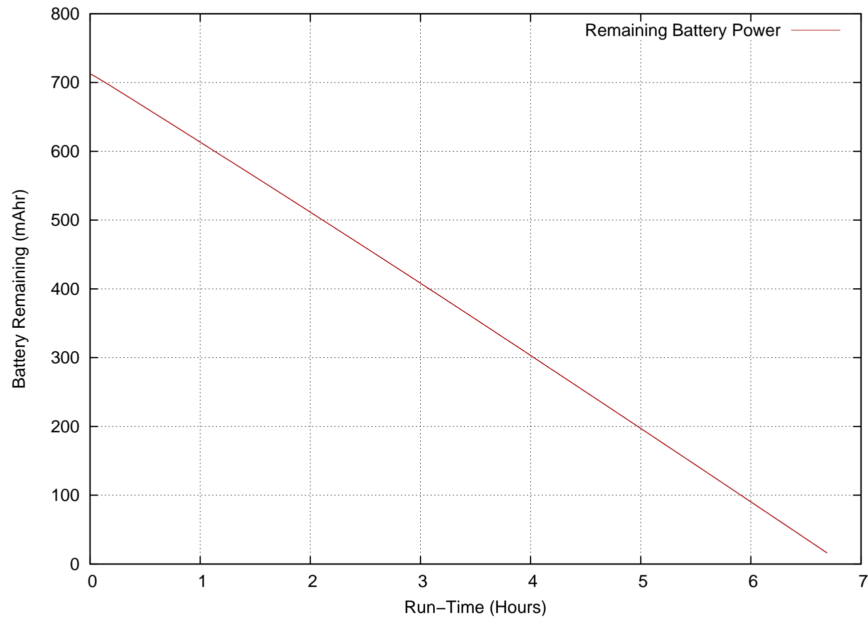


Fig. 5.7: Battery discharge while running software.

other tests, and operating under ideal conditions, i.e., initially fully-charged, and placed next to the base-station to minimize radio signal attenuation. An evaluation of the difference between the two configurations demonstrated no statistically significant difference in power consumption, since the processor and radio were both in constant use, within each configuration.

Overall, the rate of battery discharge yielded a figure of 6.91 hours as being the expected operational lifetime of a Sun SPOT mote that is continually measuring, analysing and then reporting the various on-board sensors (as well as simultaneously performing some timing analysis on the same process). Although this operational lifetime is very small compared to the operational expectations within typical WSNs [Akyildiz et al., 2002], a real-world deployment of Sun SPOT motes would be unlikely to have active sensor analysis on-going at all times, i.e., low-powered sleep periods would be used to prolong the operational lifetime of the mote, possibly triggered by periods of reduced variation within sensor readings.

This inherent limitation on the power supply of the mote may provide a motivating reason for both limited local sensor processing, as well as time-triggered dynamic software adaptation. In order to conserve batter power, the sensor function may periodically sample the environment, and then enter a low-powered sleep mode for a pre-determined period. Once enough sensor

readings have been collected, the data is analyzed locally, and a summary report transmitted. If the execution time of the sensor analysis is insufficient within a rapidly changing operating environment, an exceedance of the established WCET or ACET timing estimates can be used to trigger an adaptation of the software in order to provide a faster response.

5.4 Off-line Analysis

The experimental scenario provided a number of useful evaluations of the expected accuracy and operational overhead of TimePredict, testing its performance on a resource-constrained device within a live operating environment. To further analyze the capabilities of the TimePredict approach, two additional off-line assessments were considered;

- A comparative analysis of the accuracy and precision of TimePredict against a statically-derived worst-case and average-case timing estimate, based on the same set of timing measurement data.
- An analysis of the effects of different lag lengths and array sizes on the predictive performance, using a series of simulated timing measurements.

The former analysis evaluates the predictive performance of the TimePredict approach against an 'ideal' timing analysis, performed off-line, with access to pre-existing timing measurement data, and not subject to the periodic functional adaptations that are typically encountered within dynamically-adaptable systems. The latter analysis adjusts the internal parameters within TimePredict, to evaluate the effects of using relatively more or less timing measurement data to generate timing estimates. The aim of these two evaluations is to establish the efficacy of TimePredict as an execution-time forecasting method in its own right, as well as determine of its optimum configuration.

5.4.1 Comparative Off-line Statistical Analysis

In order to compare the predictive performance of TimePredict against an equivalent off-line timing analysis, the measurements captured during the initial run-time testing process were re-used as the basis for a static average-case and worst-case timing analysis. The aim of this static timing analysis was to demonstrate the predictive performance that can be achieved under ideal

circumstances, i.e., having a pre-existing set of timing measurements, unlimited time to undertake the analysis and manual supervision of the overall process. By contrasting the performance of the TimePredict estimates against off-line generated timing forecasts, the relative accuracy and precision of the run-time predictive models may be revealed.

	Average-Case				Worst-Case			
	Off-line		TimePredict		Off-line		TimePredict	
Config. No.	Acc. (%)	Prec. (ms)	Acc. (%)	Prec. (ms)	Acc. (%)	Prec. (ms)	Acc. (%)	Prec.. (ms)
1	100.00	2.0	99.36	4.15	100.00	0.3	100.00	11.58
2	99.50	3.0	98.91	4.18	100.00	4.2	99.91	8.17
3	100.00	3.0	99.22	5.50	100.00	0.7	99.90	10.33
4	98.00	3.0	99.15	5.67	100.00	7.9	99.96	11.42
5	98.50	3.0	98.87	4.44	100.00	5.1	99.85	7.71
6	97.50	4.0	99.29	7.51	100.00	9.9	99.92	14.28
7	99.50	5.0	98.51	6.54	100.00	10.2	99.98	20.13
8	97.00	38.9	95.10	27.76	99.50	43.4	99.83	55.79
9	100.00	0.0	99.50	2.46	100.00	0.0	100.00	11.65
10	100.00	1.0	99.49	3.14	100.00	1.0	99.86	12.04
11	100.00	1.0	99.85	3.96	100.00	1.3	99.80	9.77
12	98.00	5.9	97.12	11.06	100.00	20.7	99.80	32.37
13	99.00	0.0	99.01	3.34	100.00	3.0	99.88	10.47
14	97.00	6.9	98.10	12.78	99.50	13.6	99.65	13.91
15	94.50	18.8	98.08	24.26	99.00	25.8	99.46	18.76
16	94.50	12.0	97.85	27.55	100.00	21.8	98.85	22.38
All (Ave.)	98.31	6.72	98.59	10.06	99.88	10.56	99.79	16.92

Table 5.11: Accuracy and precision of an equivalent off-line timing analysis.

The more accurate of the worst-case models used within TimePredict was the GEV model, and consequently it was used for this analysis since it displayed a better overall accuracy and precision

than its heuristic-based counterpart. The value used for the GEV model was taken from Table 5.5, whereas the accuracy and precision of the off-line timing analysis was calculated using the MiniTab statistical package. The timing measurements of the sensor analysis function, recorded at run-time and logged by the Adaptation Server, were used as the basis for the off-line generated timing estimates. Separate analysis and validation datasets were created by partitioning the logged timing measurements into two equal-sized sets of 1,000 measurements each. Since the off-line analysis tools usually applied to the generation of WCET or ACET measurements did not support any bytecode simulation on the Squawk JVM, an equivalent statistical analysis was applied to extrapolate a worst-case and average-case timing estimate from the available data. A statistical model of the software timeliness was generated using the first (analysis) dataset, creating a worst-case bound using the maximum observed timing measurement, the mean, and the standard error of the mean. Similarly, the off-line average-case estimates were created by generating timing bounds to encapsulate the middle 95% of the same dataset, i.e., the percentile values that encapsulate 2.5% and 97.5% of the collected timing data. The second (validation) dataset was then used to test the average-case and worst-case estimates, and provide the performance metrics used for a comparative analysis with TimePredict. If the validation dataset was a mirror of the analysis dataset, the expected accuracy would be 95% and 100% for the static average-case and worst-case estimates respectively. Since these static estimates cannot automatically re-adjust their timing bounds based on adaptations occurring within the software, the timing measurements for each configuration of software were manually separated, and individually analyzed, with the predictive performance of this static analysis presented in Figure 5.11.

The predictive accuracy and precision of both TimePredict and a comparative static timing analysis is presented in Table 5.11. On initial inspection, it seems the accuracy of the TimePredict forecasting models can be compared favourably to the results of the off-line analysis. The accuracy of the average-case estimates generated by TimePredict surpass the static timing estimates in 7 out of the 16 configurations of the software, and have a mean accuracy of 98.59% compared to the statically-derived 98.31%. The slightly better performance of the average-case model compared to the off-line estimates may be ascribed to the reactive nature of the model, i.e., subtle trends or variations within the measurement data are used to modify the smoothing parameters that in turn affect the timing bounds. Within the off-line estimate, no possible alter-

ation of the bounds is possible during run-time, so that increases in variation tend to decrease the overall accuracy of the estimate. In comparison, the worst-case estimates are only marginally less accurate than their off-line generated equivalents overall (99.79% compared to 99.88%), but are more accurate within 3 of the 16 configurations of the system.

The improvement in the overall average-case accuracy due to TimePredict was 0.28% compared to the off-line analysis, with the worst-case estimates being only 0.09% less accurate, i.e., a difference of 1 in every 1,110 estimates. However, the off-line generated timing estimates show a higher overall precision across the 16 configurations of the system, with an overall difference in precision of 3.34ms and 6.36ms compared to TimePredict's average-case and worst-case forecasts respectively. This indicates that the estimates produced by TimePredict at run-time are only slightly below what can be achieved using a static timing analysis method executed off-line under ideal circumstances. The extent of these differences are minimal given the operational constraints involved in the TimePredict forecasting process, i.e., limited timing data available, periodic adaptations to the software functionality, a resource-restricted operating environment and a timing analysis process that executes concurrently with the software being analyzed. However, although the accuracy and precision of the TimePredict estimates are marginally less than their off-line equivalents, they still achieve their minimum expected accuracy requirements of 95% for average-case estimates and 99% for worst-case estimates. In summary, it can be asserted that the forecasting capabilities of the TimePredict approach are comparable to the results that could be achieved under ideal conditions, even though the estimates themselves are produced on a resource-constrained embedded device and generated at run-time.

5.4.2 Effect of Model Parameters on Predictive Performance

Both the ACET and WCET datasets are restricted to a finite number of timing measurements, and due to the constraints of the operating environment, are typically dataset on the order of several hundred measurements or less. However, by selectively adding only large timing measurements (in the case of WCET) or replacing older values with more recent measurements (in the case of ACET), more representative measurement data can be stored for subsequent run-time analysis. The number of timing measurements stored within the system effectively limits the analysis to a pre-determined sample size. However, since the models used within TimePredict may be sensitive to changes in this sample size, an analysis of the effects of storing relatively

more or fewer measurements may prove salutary. Similarly, the more measurements retained by TimePredict for analysis purposes, the greater the impact of this analysis on the underlying system resources (e.g., CPU and memory). By evaluating the effects of sample size on predictive performance, the TimePredict approach may be more easily configured to provide a suitable balance between predictive performance and the usage of limited system resources.

5.4.3 Timing Feedback

The execution time of a dynamically-adaptable system provides useful context information about its processing capabilities within the current operating environment. By predicting the timeliness of the software, a dynamically-adaptable system can determine whether functional adaptations are required to preempt any significant deterioration in its overall timing behaviour. Although this thesis is not directly concerned with the adaptation selection process for dynamically-adaptable systems, accurate timing estimates may assist in selecting, scheduling and initiating the most appropriate functional adaptation at any given time, in order to exploit changes in the prevailing operating conditions.

The initial constraints for the evaluation of TimePredict precluded a purely time-driven adaptation scenario, since a set number of measurements were required to perform a statistical analysis on its predictive accuracy and precision within each configuration of the system. However, within a live operating environment it is envisaged that adaptations will be rarely occurring events, being triggered in the main by unforeseen or excessive differences between the expected and observed functional and non-functional behaviours of the system.

The average-case timing estimates provided by TimePredict allow an ongoing assessment of the small-scale systematic timing variation within the system, whereas the worst-case estimates can be used to signal when a more fundamental change has occurred. The evaluation of TimePredict (Section 5.3.2) set a worst-case timing threshold to an artificially low value after a specified number of iterations, to prompt an adaptation to the next configuration of the system. Although this provided a very limited adaptation initiation mechanism, the subsequent accuracy and precision of the TimePredict approach showed that a well-chosen average-case or worst-case timing threshold could provide the basis for a time-optimizing dynamically-adaptable system. Indeed, TimePredict could be used to both select an appropriate adaptation threshold during run-time, as well as signify the likelihood of this threshold being exceeded based on the current

timing behaviour of the system.

Whereas the evaluation of TimePredict presented in this chapter has shown that highly-accurate timing measurements can be created at run-time, with minimal cost to the underlying system, its application as a feedback mechanism within dynamically-adaptable systems must be considered further. The potential future application of run-time timing estimates as feedback into the adaptation process is discussed in the next chapter.

5.5 Summary

This chapter has evaluated the performance of the TimePredict approach, both in terms of its predictive capabilities running on a resource-constrained embedded sensor device, as well its impact on the normal operations of the system. It has been shown that an accurate reactive timing analysis process, that uses measurements of the executing software, can be deployed successfully within a resource-constrained operating environment susceptible to periodic functional adaptations. Although this timing analysis process incurs an operational overhead on the underlying system, specifically in terms of additional memory usage, the overall effect on the system is typically no more than several hundred kilobytes of additional memory consumed. Given that the experimental scenario was configured to use all four predictive models in parallel in a continuous measurement and analysis cycle, the memory overhead may be reduced when deployed to a live environment with a less aggressive analysis interval.

The predictive accuracy and precision of the run-time timing analysis process was evaluated against a statistically-based off-line analysis, and shown to be comparable, or in some cases better. The off-line analysis must be supervised manually, and tested separately against each configuration of the dynamically adaptable system. In contrast, TimePredict generates estimates automatically at run-time, based on timing measurements taken from the executing system, updating these estimates automatically immediately following an adaptation to the software

Chapter 6

Conclusion

"θάλαττα! θάλαττα!"
"The Sea! The Sea!"
Xenophon

This thesis described TimePredict, a reactive timing analysis approach, suitable for dynamically-adaptable systems executing on resource-constrained hardware platforms. This chapter reviews the most significant achievements of the TimePredict approach, and assesses its contribution to the state of the art in the domain of software execution-time analysis. Lastly, this chapter concludes with a discussion of the remaining open research issues that may lend themselves to future work.

6.1 Contribution

Current software timing analysis techniques are limited in many ways. For example, most tool-based timing analysis approaches restrict the underlying hardware to a very basic processor architecture, since more modern (desktop) processors prove very difficult to model as a cycle-accurate representation of the hardware within software [Wilhelm et al., 2008]. Tool-based timing analysis approaches are performed exclusively off-line, under manual supervision, and using rigorous instruction-accurate simulations of the software running on a model of the target hardware platform [Sehlberg et al., 2006]. This type of systematic analysis is required, since

the effect of an incorrect timing estimate may be quite disproportional to the error, e.g., system failure, loss of life, etc. Although very precise, and sufficiently accurate for hard real-time systems, subsequent functional alterations to the software are proscribed, in order to avoid invalidating the statically-derived timing estimate. Consequently, the static timing analysis techniques used within hard real-time systems do not readily lend themselves to the evaluation of dynamically-adaptable software, even if the number and functional scope of potential configurations of the system could be known in advance (which is often not the case).

Similarly, traditional measurement-based timing analysis techniques either require a pre-generated series of test-cases to expose the various latent timing behaviours within the software [Colmenares et al., 2008], or rely on a prohibitively large number of measurements to create sufficiently accurate timing estimates [Edgar, 2002]. Within a dynamically-adaptable system, the current configuration of the software can change unexpectedly, requiring updated timing estimates immediately following every adaptation. However, a lengthy timing analysis process may lead to periods where no valid timing estimates are available for the newly adapted software, which can in turn negate any benefits that might be accrued through run-time functional adaptations. For example, within closely-coupled systems, the outputs of one part of the system may form the inputs of another, such that small changes in the timeliness of the software can propagate through the system, leading to missed timing deadlines, reduced throughput or even functional errors within otherwise dependable code. Conversely, a prompt accurate assessment of the likely execution time of dynamically-adaptable software can reduce uncertainties about its timing behaviour and minimize the possibility of time-induced functional errors.

The TimePredict approach was designed for dynamically-adaptable systems operating with soft real-time constraints, i.e., systems where occasionally exceeding the estimated timing behaviour results in a drop in the overall quality of service rather than a complete system failure. Since these systems can modify their functionality as well as their timing behaviour during run-time, TimePredict must both generate estimates using very little available information, and react to sudden changes in software timeliness caused by functional adaptations. Specialized statistical models are used to forecast the execution time of the software, using the limited timing data available immediately following an adaptation. TimePredict uses both heuristic-based and model-based forecasting methods, the former to provide usable estimates in cases where there is limited timing measurement data, and the latter for when more information has been col-

lected about the system. The accuracy and precision of the various forecasting methods used within TimePredict were assessed using the scenario of a dynamically-adaptable sensor analysis function executing on a resource-constrained Java Sun SPOT mote. While being tested within this challenging operating environment, TimePredict generated approximately 32,000 timing estimates, and was found to meet its expected accuracy of 95% and 99% for the average-case and worst-case execution times respectively. In addition, an off-line timing analysis was performed on the same data under ideal circumstances, and the results were found to compare favourably in both accuracy and precision to those of TimePredict, even though the latter estimates were produced automatically at run-time within a dynamically-adaptable system.

The limitations of TimePredict can be summarized within the context and constraints of its operating environment. Although each predictive model used by TimePredict was selected to fulfill a specific role, within a particular operational scenario, the additional overhead caused by maintaining four separate predictive models could be reduced. For example, within systems with very stable timing behaviours, executing multiple predictive models in parallel unnecessarily consumes systems resources. This additional overhead may be eased by using only one of the two types of predictive models (ACET/WCET), depending on the requirements of the system at a given time. For example, where correct task scheduling takes precedence, the worst-case estimates can be generated exclusively, whereas average-case estimates may be more appropriate within environments that place a greater emphasis on QoS considerations.

Although the estimates produced by TimePredict are sufficiently accurate within the loose timing constraints present within soft real-time systems such as WSNs, this level of accuracy is insufficient within more hard real-time environments. TimePredict assumes no off-line analysis is possible on the underlying dynamically-adaptable system, i.e., its functional makeup is completely fluid and indeterminate from within an off-line context. However, there may be scope to improve the overall accuracy of the system by incorporating some limited off-line analysis on parts of the system, or re-using previous timing data taken from comparable configurations within the same system. In addition, TimePredict could be extended to extrapolate between different configurations of the same system within similar operating conditions, to select the predictive models that may provide the highest accuracy for the current operating environment.

However, predictive accuracy was not the only consideration when evaluating TimePredict - it must also limit the impact of the timing analysis process on the underlying system. Since

resource-constrained devices such as wireless sensor motes can be easily over-loaded if used incorrectly, TimePredict was designed to minimize the use of system resources such as memory and processor cycles required to generate timing estimates. An evaluation of the impact of TimePredict showed that it caused no disruption to the normal timeliness of the software when executed at preset intervals, while only consuming somewhere between 69kB and 230kB of memory. If the timing analysis methods employed by TimePredict can achieve their stated levels of accuracy and precision within a dynamically-adaptable system, executing on a wireless sensor mote with a 180MHz processor and only 512kB of RAM, it can rightfully claimed that the TimePredict approach has fulfilled its goals.

6.2 Future Work

As is the case within all research, particularly within studies that have few direct antecedents, the limited time available entails that some topics remain open to further investigation, both in the short term and over a longer period [Howlin, 1994].

Although TimePredict was evaluated using a single dynamically-adaptable sensor mote, its deployment within a large-scale WSN could provide a number of interesting real-time applications. The use of timing estimates as feedback into an adaptation selection process, could be further tested with respect to the overall optimization of mote software throughout the network. This timing feedback could be augmented with timing measurements taken from other network-connected motes, such that each mote looks to its similarly-configured neighbours for additional timing measurement data, or adaptation advice. Although TimePredict is solely focused on forecasting the execution time behaviour within a single mote, the predictive models could be distributed throughout a WSN, and applied to forecasting network-wide phenomena, such as the end-to-end transmission time of messages or the effects of adaptation-related down-time on the operation of the remaining unadapted motes.

The type of dynamically-adaptable WSNs considered within this thesis had each mote maintaining complete control over the selection and initiation of adaptations. However, instead of motes autonomously adapting their behaviour to suit local conditions, another type of dynamically-adaptable WSN may delegate the responsibility for adaptations to a single node within the network. In this scenario, the controlling node would select a particular software adaptation, coercing other nodes within the network to enact same functional changes based on previously

reported information. In this network-wide adaptation, the propagation of the adaptations themselves will be of importance within the WSN, especially if node restarts are required to bring the mote back up with the new configuration of the software. In this case, the timeliness of the adaptation process, rather than the software itself, may need to be predicted, in order to guarantee the correct roll-out of a network-wide functional change.

Lastly, the performance of the TimePredict approach can be further investigated, especially its distribution selection and model-fitting process. A heuristic-driven rather than heuristic-based analysis method might be better used to quickly determine the optimal settings for TimePredict itself, such as selecting the appropriate lag length, the most likely worst-case model, and the optimal analysis interval. Currently, the TimePredict approach does not perform any pattern analysis between the timing behaviours of different configurations of a dynamically-adaptable system. However, the integration of a number of different AI-based pattern matching techniques might be able to expose some facets of the timing behaviour common across all the configurations of a particular system, perhaps by examining the levels of co-variance between software timeliness, and system-level properties such as context-switches, memory usage, I/O latencies, or the number of concurrent processes.

6.3 Conclusion

This chapter reviewed the motivations for TimePredict, and its achievements in predicting the timeliness of dynamically-adaptable software within highly restrictive operating environments. In particular the approach taken by TimePredict demonstrates that accurate forecasts of software timing behaviour are possible using a reactive measurement-based timing analysis process, without excessively impacting on the underlying system. Finally, this chapter concluded with a description of the possible areas for future research within the domain of measurement-based timing analysis for resource-constrained systems.

Appendix A

Software Execution Times

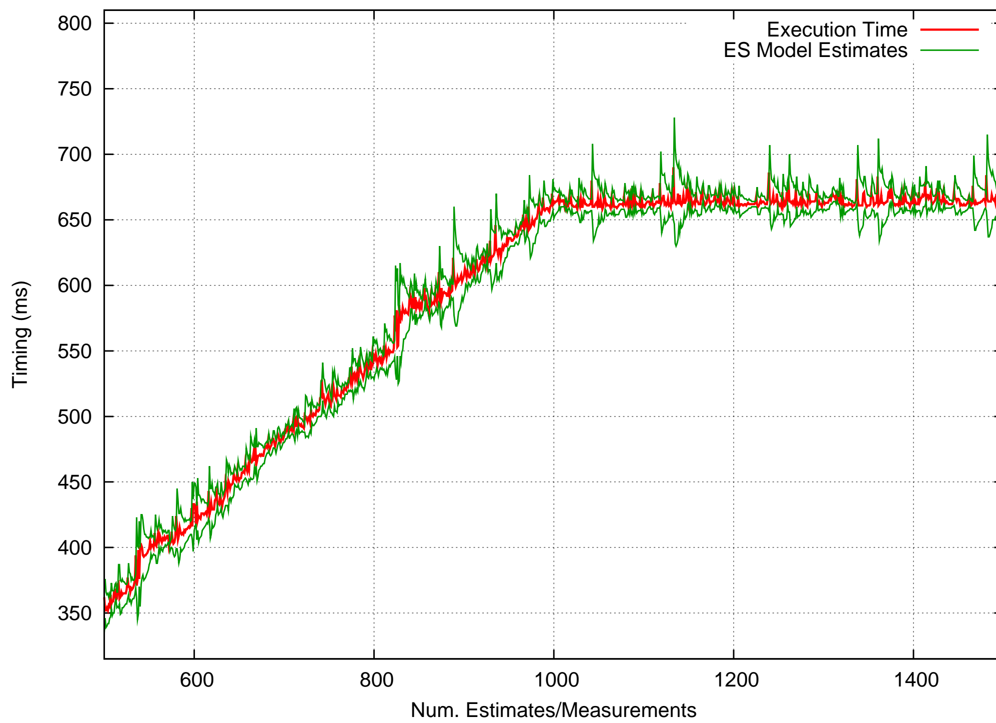


Fig. A.1: Detail of the ACET Timing Estimates for Configuration 16, with the red line representing execution time behaviour and the green lines representing the ES estimate bounds.

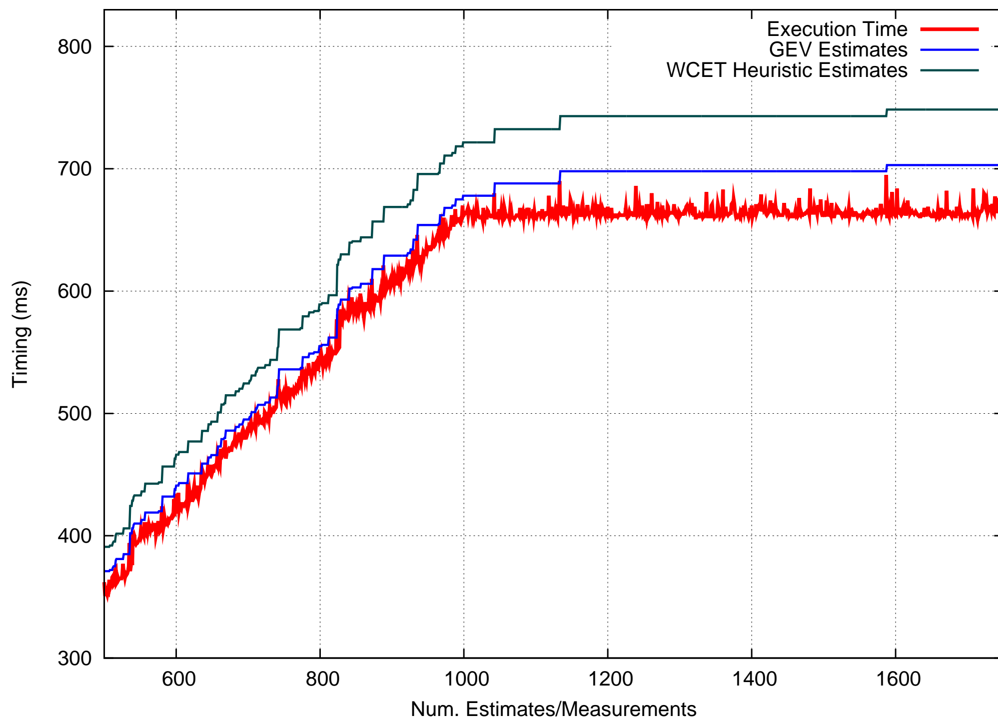
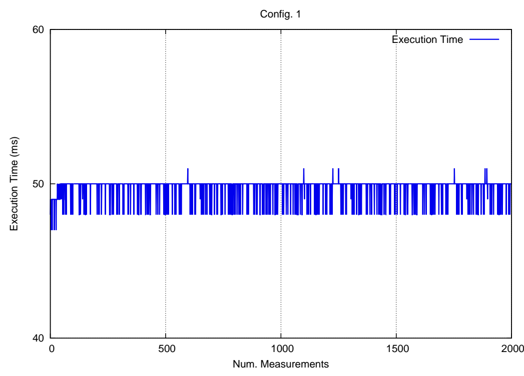
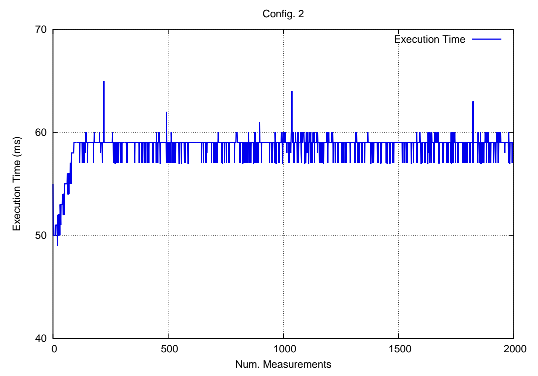


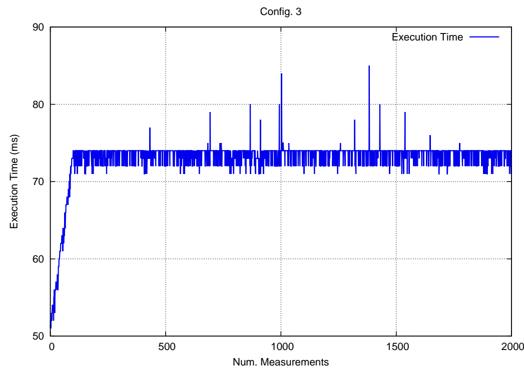
Fig. A.2: The same timing behaviour for Configuration 16, overlaid with worst-case estimate bounds.



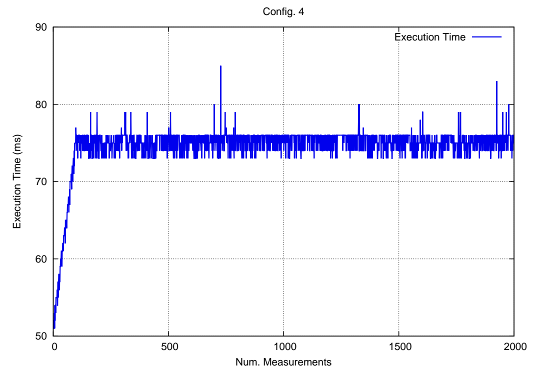
(a) Configuration 1



(b) Configuration 2

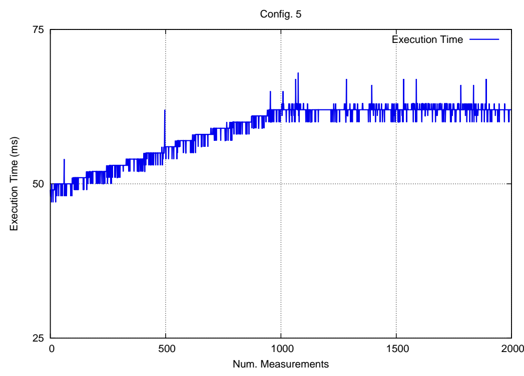


(c) Configuration 3

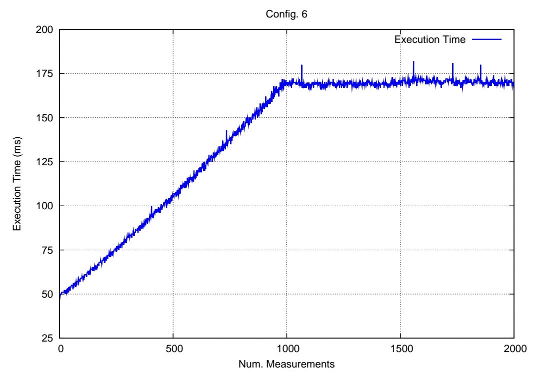


(d) Configuration 4

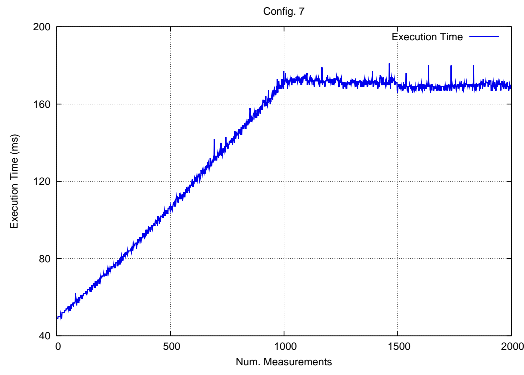
Fig. A.3: Execution Times of Configurations 1 to 4



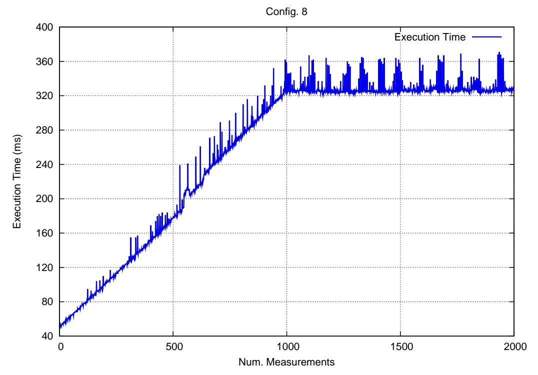
(a) Configuration 5



(b) Configuration 6

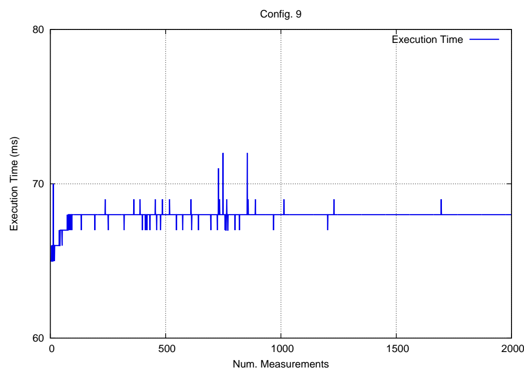


(c) Configuration 7

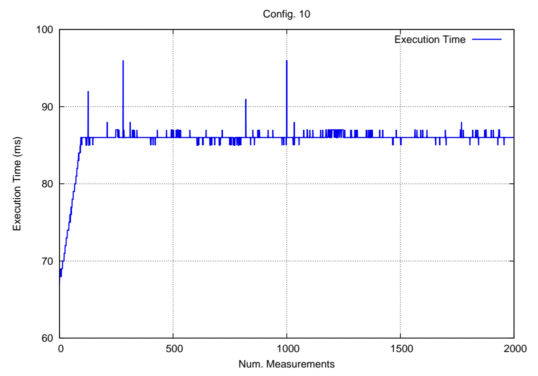


(d) Configuration 8

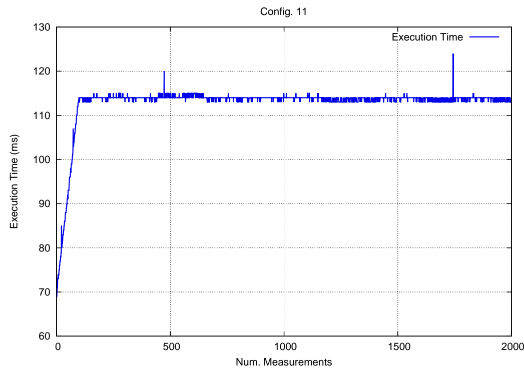
Fig. A.4: Execution Times of Configurations 5 to 8



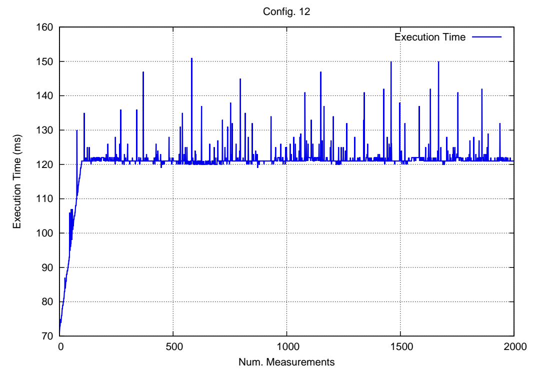
(a) Configuration 9



(b) Configuration 10

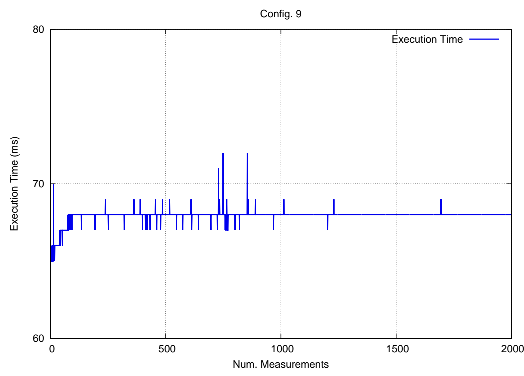


(c) Configuration 11

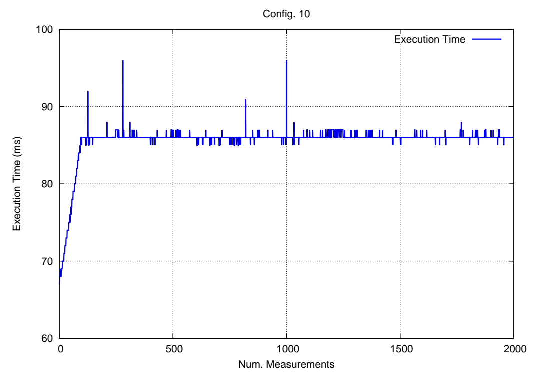


(d) Configuration 12

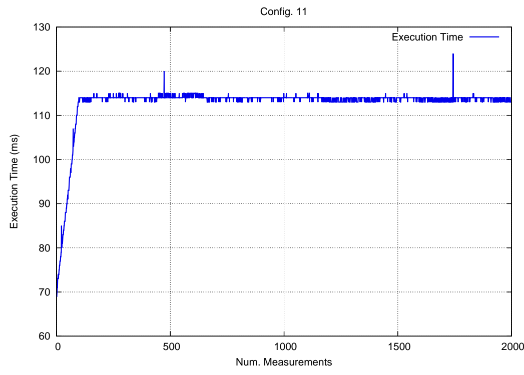
Fig. A.5: Execution Times of Configurations 9 to 12



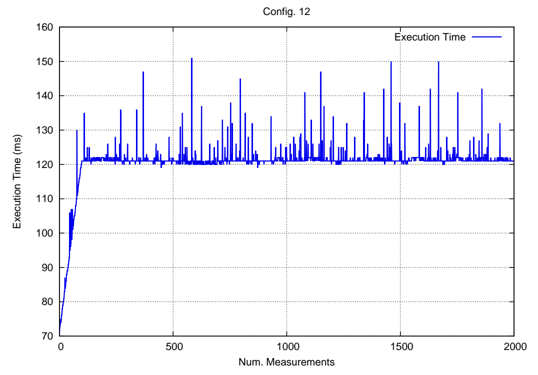
(a) Configuration 13



(b) Configuration 14



(c) Configuration 15



(d) Configuration 16

Fig. A.6: Execution Times of Configurations 13 to 16

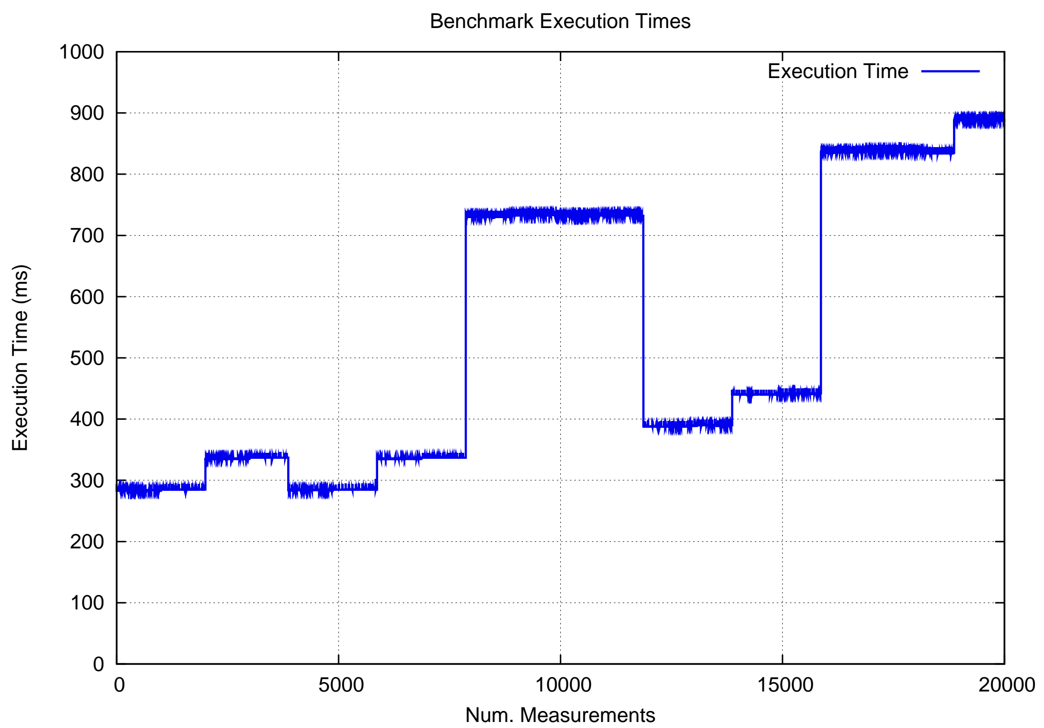


Fig. A.7: The timing performance resulting from the execution of the benchmark functions. This dataset is used to evaluate TimePredict off-line.

Bibliography

[Adler et al., 2007] Adler, R.; Schneider, D.; and Trapp, M. (2007). "Development of Safe and Reliable Embedded Systems using Dynamic Adaptation". In *Workshop on Model-Driven Software Adaptation (M-ADAPT'07) in conjunction with ECOOP'07*, pages 9–14.

[Akyildiz et al., 2002] Akyildiz, I. F.; Su, W.; Sankarasubramaniam, Y.; and Cayirci, E. (2002). "Wireless sensor networks: a survey". *Computer Networks*, 38(4), pp. 393–422.

[Alvarado et al., 1998] Alvarado, E.; Sandberg, D. V.; and Pickford, S. G. (1998). "Modeling large forest fires as extreme events". *Northwest Science*, 72, pp. 66–75.

[Arndt et al., 2009] Arndt, H.; Bundschus, M.; and Naegele, A. (2009). "Towards a Next-Generation Matrix Library for Java". In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 460–467.

[Arnold et al., 2005] Arnold, K.; Gosling, J.; and Holmes, D. (2005). *The Java Programming Language*. Addison-Wesley, 4th edition.

[Arseneau et al., 2006] Arseneau, E.; Goldman, R.; Poursohi, A.; Smith, R. B.; and Daniels, J. (2006). "Simplifying the Development of Sensor Applications". In *OOPSLA Workshop on Building Software for Sensor Networks (BSSN'06)*.

[Asikainen et al., 2003] Asikainen, T.; Soininen, T.; and Männistö, T. (2003). "A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families". In *Proceedings of the 13th International Software Product Line Conference*, pages 225–249.

[Assaf and Noyé, 2008] Assaf, A. and Noyé, J. (2008). "Dynamic AspectJ". In *In Proceedings of the 2008 symposium on Dynamic Languages (DLS'08)*, pages 1–12.

[Ayed and Berbers, 2007] Ayed, D. and Berbers, Y. (2007). "Dynamic adaptation of CORBA component-based applications". In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 580–585.

[Barrenetxea et al., 2008] Barrenetxea, G.; Ingelrest, F.; Schaefer, G.; and Vetterli, M. (2008). "The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments". In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SENSYS'08)*, pages 43–56.

[Batista et al., 2005] Batista, T.; Joolia, A.; and Coulson, G. (2005). "Managing Dynamic Reconfiguration in Component-Based Systems". In *Proceedings of the European Workshop on Software Architectures*, pages 1–18.

[Becker et al., 2006] Becker, S.; Grunske, L.; Mirandola, R.; and Overhage, S. (2006). "Performance Prediction of Component-Based Systems - A Survey from an Engineering Perspective". *Architecting Systems With Trustworthy Components*, 3938, pp. 169–192.

[Becker et al., 2009] Becker, S.; Koziolek, H.; and Reussner, R. (2009). "The Palladio Component Model for Model-driven Performance Prediction". *Journal of Systems and Software*, 82(1), pp. 3–22.

[Beltrame et al., 2001] Beltrame, G.; Brandolese, C.; Fornaciari, W.; Salice, F.; Sciuto, D.; and Trianni, V. (2001). "Dynamic Modeling of Inter-Instruction Effects for Execution Time Estimation". In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*, pages 136–141.

[Bernat et al., 2003] Bernat, G.; Colin, A.; and Petters, S. (2003). "pWCET: a Tool for Probabilistic WCET Analysis of Real-Time Systems". Technical Report YCS-2003-353, Department of Computer Science, University of York, UK.

[Bhylin et al., 2005] Bhylin, S.; Ermedahl, A.; Gustafsson, J.; and Bjorn, L. (2005). "Applying Static WCET Analysis to Automotive Communication Software". In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 249–258.

[Biyani and Kulkarni, 2005] Biyani, K. N. and Kulkarni, S. S. (2005). "Building component families to support adaptation". In *Proceedings of the 2005 ICSE workshop on Design and Evolution of Autonomic Application Software (DEAS'05)*, pages 1–7, New York, NY, USA. ACM.

[Bruneton et al., 2006] Bruneton, E.; Coupaye, T.; Leclercq, M.; Quéma, V.; and Stefani, J.-B. (2006). "The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems". *Software - Practice and Experience*, 36(11-12), pp. 1257–1284.

[Buisson et al., 2005] Buisson, J.; André, F.; and Pazat, J.-L. (2005). "A Framework for Dynamic Adaptation of Parallel Components". In *Proceedings of the International Conference on Parallel Computing (ParCo 2005)*, pages 65–72, Malaga, Spain.

[Buisson et al., 2007] Buisson, J.; Andre, F.; and Pazat, J.-L. (2007). "Supporting adaptable applications in grid resource management systems". In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID'07)*, pages 58–65.

[Burguiére and Rochange, 2006] Burguiére, C. and Rochange, C. (2006). "History-based Schemes and Implicit Path Enumeration". In *6th International Workshop on Worst-Case Execution Time (WCET) Analysis*.

[Cadenas and Rivera, 2010] Cadenas, E. and Rivera, W. (2010). "Wind speed forecasting in three different regions of Mexico, using a hybrid ARIMA-ANN model". *Renewable Energy*, 35(12), pp. 2732–2738.

[Caicedo, 2006] Caicedo, A. (2006). "The Sun Small Programmable Object Technology (Sun Spot)". Sun Tech Days 2006-2007. Enter text here.

[Calinescu and Kwiatkowska, 2009] Calinescu, R. and Kwiatkowska, M. (2009). "Using quantitative analysis to implement autonomic IT systems". In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE'09)*, pages 100–110, Vancouver, Canada.

[Castilloa et al., 2006] Castilloa, E.; Lopez-Aenlle, M.; Ramos, A.; Fernandez-Canteli, A.; Kieselbach, R.; and Esslinger, V. (2006). "Specimen length effect on parameter estimation in modelling fatigue strength by Weibull distribution". *International Journal of Fatigue*, 28(9), pp. 1047–1058.

[Cervin et al., 2003] Cervin, A.; Henriksson, D.; Lincoln, B.; Eker, J.; and Arzen, K.-E. (2003). "How Does Control Timing Affect Performance?". *IEEE Control Systems Magazine*, 23(3), pp. 16–30.

[Chambers, 2008] Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*. Springer, 2nd edition.

[Chatfield, 2003] Chatfield, C. (2003). *The Analysis of Time Series: An Introduction*. Chapman and Hall/CRC, 6th edition.

[Chen et al., 2001] Chen, K.; Malik, S.; and August, D. I. (2001). "Retargetable static timing analysis for embedded software". In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*, pages 39–44.

[Chen et al., 2008] Chen, L.; McKerrow, P.; and Lu, Q. (2008). "Developing Real-time Applications with Java Based Sun SPOT". In *Proceedings of the 2008 Australasian Conference on Robotics and Automation*.

[Cheng et al., 2008] Cheng, B. H.; Giese, H.; Inverardi, P.; Magee, J.; and et al. (2008). "Software Engineering for Self-Adaptive Systems: A Research Road Map". In *Software Engineering for Self-Adaptive Systems*, number 08031 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Chess et al., 2003] Chess, D. M.; Palmer, C. C.; and White, S. R. (2003). "Security in an autonomic computing environment". *IBM Systems Journal*, 42(1), pp. 107–118.

[Childs et al., 2006] Childs, A.; Greenwald, J.; Jung, G.; Hoosier, M.; and Hatcliff, J. (2006). "Calm and Cadena: Metamodeling for component-based product-line development". *Computer*, 39, pp. 42–50.

[Chou, 2005] Chou, R. Y.-T. (2005). "Forecasting Financial Volatilities with Extreme Values: The Conditional Autoregressive Range (CARR) Model". *Journal of Money, Credit, and Banking*, 37(3), pp. 561–582.

[Coles, 2001] Coles, S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer, 1st edition.

[Colin and Puaut, 2001] Colin, A. and Puaut, I. (2001). "Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System". In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, pages 191–198.

[Colmenares et al., 2008] Colmenares, J. A.; Chansik, I.; Kim, K.; Klefstad, R.; and Chae-Deok, L. (2008). "Measurement Techniques in a Hybrid Approach for Deriving Tight Execution-time Bounds of Program Segments in Fully-featured Processors". In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 68–79.

[Cooper et al., 2002] Cooper, K. D.; Subramanian, D.; and Torczon, L. (2002). "Adaptive Optimizing Compilers for the 21st Century". *The Journal of Supercomputing*, 23(1), pp. 7–22.

[Corsaro and Schmidt, 2002] Corsaro, A. and Schmidt, D. C. (2002). "Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems". In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 90–100.

[Costa et al., 2007] Costa, P.; Coulson, G.; Gold, R.; Lad, M.; Mascolo, C.; Mottola, L.; Picco, G. P.; Sivaharan, T.; Weerasinghe, N.; and Zachariadis, S. (2007). "The RUNES middleware for networked embedded systems and its application in a disaster management scenario". In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications (Percom'07)*, pages 69–78.

[Deverge and Puaut, 2005] Deverge, J.-F. and Puaut, I. (2005). "Safe measurement-based WCET estimation". In *Proceedings of 5th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*.

[Dhurjati et al., 2006] Dhurjati, D.; Kowshik, S.; and Adve, V. (2006). "SAFECode: enforcing alias analysis for weakly typed languages". *ACM SIGPLAN Notices*, 41(6), pp. 144–157.

[Diaconescu and Murphy, 2005] Diaconescu, A. and Murphy, J. (2005). "Automating the Performance Management of Component-Based Enterprise Systems through the use of Redundancy". In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 44–53, Long Beach, CA, USA.

[Dobson et al., 2006] Dobson, S.; Denazis, S.; Fernández, A.; Gaïti, D.; Gelenbe, E.; Massacci, F.; Nixon, P.; Saffre, F.; Schmidt, N.; and Zambonelli, F. (2006). "A survey of autonomic communications". *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2), pp. 223–259.

[Dowling and Cahill, 2001] Dowling, J. and Cahill, V. (2001). "Dynamic Software Evolution and the K-Component Model". In *Workshop on Software Evolution, at OOPSLA 2001.*

[Dyer and Rajan, 2008] Dyer, R. and Rajan, H. (2008). "Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation". In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 191–202.

[Edgar, 2002] Edgar, S. (2002). *Estimation of worst-case execution time using statistical analysis, PhD thesis.* PhD thesis, Department of Computer Science, University of York.

[Edwards and Lee, 2007] Edwards, S. A. and Lee, E. A. (2007). "The Case for the Precision Timed (PRET) Machine". In *Proceedings of the 44th Annual Conference on Design Automation*, pages 264–265.

[Engblom et al., 2001] Engblom, J.; Ermedahl, A.; Sjodin, M.; Gustafsson, J.; and Hansson, H. (2001). "Applying Static WCET Analysis to Automotive Communication Software". *International Journal of Software Tools for Technology Transfer*, 4, pp. 437–455.

[Engle, 1982] Engle, R. F. (1982). "Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of U.K. Inflation". *Econometrica*, 50, pp. 987–1008.

[Ensink et al., 2003] Ensink, B.; Stanley, J.; and Adve, V. (2003). "Program Control Language: a programming language for adaptive distributed applications". *Journal of Parallel and Distributed Computing*, 63(11), pp. 1082–1104.

[Epifani et al., 2009] Epifani, I.; Ghezzi, C.; Mirandola, R.; and Tamburrelli, G. (2009). "Model evolution by run-time parameter adaptation". In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 111–121, Washington, DC, USA. IEEE Computer Society.

[Ermedahl et al., 2005] Ermedahl, A.; Gustafsson, J.; and Lisper, B. (2005). "Experiences from Industrial WCET Analysis Case Studies". In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, pages 25–28.

[Ermedahl et al., 2007] Ermedahl, A.; Sandberg, C.; Gustafsson, J.; Bygde, S.; and Lisper, B. (2007). "Loop Bound Analysis based on a Combination of Program Slicing, Abstract In-

terpretation, and Invariant Analysis". In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*.

[Eskenazi et al., 2004] Eskenazi, E.; Fioukov, A.; and Hammer, D. (2004). "Performance Prediction for Component Compositions". In *7th International Symposium on Component-based Software Engineering (CBSE'04)*, pages 280–293.

[Evans et al., 2000] Evans, M.; Hastings, N.; and Peacock, B. (2000). *Statistical Distributions*. Wiley-Interscience, 3rd edition.

[Fredriksson et al., 2007] Fredriksson, J.; Nolte, T.; Ermedahl, A.; and Nolin, M. (2007). "Clustering Worst-Case Execution Times for Software Components". In *Proceedings of the 7th international workshop on worst case execution time analysis (WCET'07)*.

[Fritsch and Clarke, 2008] Fritsch, S. and Clarke, S. (2008). "TimeAdapt: timely execution of dynamic software reconfigurations". In *Proceedings of the 5th Middleware doctoral symposium (MDS'08)*, pages 13–18.

[Gal et al., 2006] Gal, A.; Probst, C. W.; and Franz, M. (2006). "HotpathVM: an effective JIT compiler for resource-constrained devices". In *Proceedings of the 2nd international conference on Virtual execution environments (VEE '06)*, pages 144–153.

[Gardner, 1985] Gardner, E. S. (1985). "Exponential Smoothing: The State of the Art". *Journal of Forecasting*, 4(1), pp. 1–28.

[Georgiadis et al., 2002] Georgiadis, I.; Magee, J.; and Kramer, J. (2002). "Self-organising software architectures for distributed systems". In *Proceedings of the first workshop on Self-healing systems*, pages 33–38.

[Ghosal et al., 2004] Ghosal, A.; Henzinger, T. A.; Kirsch, C. M.; and Sanvido, M. A. A. (2004). "Event-Driven Programming with Logical Execution Times". In *Proceedings of the 7th International Workshop on Hybrid Systems: Computation and Control (HCCC 2004)*, pages 357–371.

[Goldman, 2007] Goldman, R. (2007). "A Sun SPOT Application Note: Using the AT91 Timer/Counter". Technical report, Sun Microsystems Inc., Santa Clara, CA, USA.

[Goldsby et al., 2008] Goldsby, H. J.; Cheng, B. H.; and Zhang, J. (2008). "AMOEBA-RT: Run-Time Verification of Adaptive Software". In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, pages 212–224. Springer Verlag.

[Guo et al., 2008] Guo, X.; Boubekeur, M.; Mc Enery, J.; and Hickey, D. (2008). "A new approach for ACET based scheduling of soft real-time systems". In *Proceedings of the 12th WSEAS international conference on Computers*, pages 886–892.

[Hansen et al., 2009] Hansen, J.; Hissam, S.; and Moreno, G. (2009). "Statistical-Based WCET Estimation and Validation". In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, pages 123–133, Dublin, Ireland.

[Henia et al., 2005] Henia, R.; Hamann, A.; Jersak, M.; Racu, R.; Richter, K.; and Ernst, W. (2005). "System Level Performance Analysis - the SymTA/S Approach". *IEE Computers and Digital Techniques*, 152(2), pp. 148–166.

[Heo and Abdelzaher, 2009] Heo, J. and Abdelzaher, T. (2009). "AdaptGuard: guarding adaptive systems from instability". In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC'09)*, pages 77–86.

[Hill et al., 2008] Hill, J. H.; Schmidt, D. C.; Porter, A. A.; and Slaby, J. M. (2008). "CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments". In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'08)*, pages 66–75, Belfast, Northern Ireland.

[Hilsdale and Hugunin, 2004] Hilsdale, E. and Hugunin, J. (2004). "Advice weaving in AspectJ". In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 26–35.

[Hinton et al., 2006] Hinton, A.; Kwiatkowska, M.; Norman, G.; and Parker, D. (2006). "PRISM: A tool for automatic verification of probabilistic systems". In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS06)*, pages 441–444, Vienna, Austria.

[Hissam et al., 2003] Hissam, S.; Moreno, G.; Stafford, J.; and Wallnau, K. (2003). "Enabling predictable assembly". *Journal of Systems and Software: Special Issue on Component-Based Software Engineering*, 65(3), pp. 185–198.

[Hissam et al., 2008] Hissam, S. A.; Moreno, G. A.; Plakosh, D.; Savo, I.; and Stelmarczyk, M. (2008). "Predicting the Behavior of a Highly Configurable Component Based Real-Time System". In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems (ECRTS'08)*, pages 57–68.

[Hoare, 1978] Hoare, C. A. R. (1978). "Communicating Sequential Processes". *Communications of the ACM*, 21(8), pp. 666–677.

[Holsti et al., 2008] Holsti, N.; Gustafsson, J.; Bernat, G.; et al. (2008). "WCET Tool Challenge 2008". In *Proceedings of 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08).*

[Howlin, 1994] Howlin, P. (1994). *Bohemian times: An outline history of Bohemian Football Club and Dalymount Park, 1890-1993.*

[Huang et al., 2008] Huang, G.; Liu, X.; and Mei, H. (2008). "Online approach to feature interaction problems in middleware based system". *Science in China Series F: Information Sciences*, 51(3), pp. 225–239.

[Hummel and Atkinson, 2010] Hummel, O. and Atkinson, C. (2010). "Automated Creation and Assessment of Component Adapters with Test Cases". In *Proceedings of the 13th International Symposium on Component Based Software Engineering (CBSE'10)*, pages 166–181.

[Ivers and Moreno, 2008] Ivers, J. and Moreno, G. A. (2008). "PACC starter kit: developing software with predictable behavior". In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 949–950.

[Keeney, 2004] Keeney, J. (2004). *Completely Unanticipated Dynamic Adaptation of Software.* PhD thesis, Department of Computer Science, Trinity College Dublin.

[Keeney and Cahill, 2003] Keeney, J. and Cahill, V. (2003). "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework". In *Proceedings of the 4th IEEE International Work-*

shop on Policies for Distributed Systems and Networks (POLICY'03), pages 3–14, Washington, DC, USA.

[Kell, 2008] Kell, S. (2008). "A Survey of Practical Software Adaptation Techniques". *J.UCS - Journal of Universal Computer Science*, 14(13), pp. 2110–2157.

[Kirner and Puschner, 2008] Kirner, R. and Puschner, P. (2008). "Obstacles in Worst-Case Execution Time Analysis". In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC'08)*, pages 333–339.

[Kniesel et al., 2001] Kniesel, G.; Costanza, P.; and Austermann, M. (2001). "JMangler - A Framework for Load-Time Transformation of Java Class Files". In *Proceedings of IEEE Workshop on Source Code Analysis and Manipulation (SCAM'01)*, pages 100–110.

[Kotz and Nadarajah, 2000] Kotz, S. and Nadarajah, S. (2000). *Extreme Value Distributions : Theory and Applications*. Imperial College Press.

[Koziolek, 2010] Koziolek, H. (2010). "Performance evaluation of component-based software systems: A survey". *Performance Evaluation*, 67(8), pp. 634–658.

[Kumar et al., 2008] Kumar, T.; Cledat, R.; Sreeram, J.; and Pande, S. (2008). "Statistically Analyzing Execution Variance for Soft Real-Time Applications". In *Proceedings 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, pages 124–140.

[Lee, 2002] Lee, E. A. (2002). "Embedded Software". *Advances in Computers*, 56, pp. 55–95.

[Li et al., 2007] Li, X.; Liang, Y.; Mitra, T.; and Roychoudhury, A. (2007). "Chronos: A timing analyzer for embedded software". *Science of Computer Programming*, 69(1-3), pp. 56–67.

[Li et al., 2010] Li, X.; Ortiz, P. J.; Browne, J.; Franklin, D.; Oliver, J. Y.; Geyer, R.; Zhou, Y.; and Chong, F. T. (2010). "Smartphone Evolution and Reuse: Establishing a more Sustainable Model". In *In Proceedings of the 2nd International Workshop on Green Computing (to appear)*.

[Lindley and Scott, 1995] Lindley, D. V. and Scott, W. F. (1995). *New Cambridge Statistical Tables*. Cambridge University Press, Cambridge, UK, 2th edition.

[Malik et al., 1997] Malik, S.; Martonosi, M.; and Li, Y.-T. S. (1997). "Static timing analysis of embedded software". In *Proceedings of the 34th annual Design Automation Conference (DAC'97)*, pages 147–152.

[Marref and Bernat, 2008] Marref, A. and Bernat, G. (2008). "Towards Predicated WCET Analysis". In *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, pages 138–147.

[Massey, 1951] Massey, F. J. (1951). "The Kolmogorov-Smirnov Test for Goodness of Fit.". *Journal of the American Statistical Association*, 46, pp. 68–78.

[McIlroy, 1968] McIlroy, D. M. (1968). "Mass Produced Software Components". In *Software Engineering; Report on a conference by the NATO Science Committee (Garmish, Germany)*, pages 138–150.

[McKinley et al., 2004] McKinley, P. K.; Sadjadi, S. M.; Kasten, E. P.; and Cheng, B. H. C. (2004). "A Taxonomy of Compositional Adaptation". Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, MI, USA.

[Menascé et al., 2004] Menascé, D. A.; Ruan, H.; and Gomaa, H. (2004). "A framework for QoS-aware software components". In *Proceedings of the 4th International Workshop on Software and Performance*, pages 186–196.

[Mezzetti et al., 2008] Mezzetti, E.; Holsti, N.; Colin, A.; Bernat, G.; and Vardanega, T. (2008). "Attacking the Sources of Unpredictability in the Instruction Cache Behavior". In *Proceedings of the 16th International Conference on Real-Time and Network Systems (RTNS'08)*, pages 151–160.

[Mills, 1990] Mills, D. L. (1990). "Network Time Protocol (Version 3) Specification, Implementation and Analysis". Technical Report 90-6-1, University of Delaware, Newark, DE, USA.

[Mindell, 2008] Mindell, D. A. (2008). *Digital Apollo: Human and Machine in Spaceflight.* MIT Press, Cambridge, MA, USA.

[Minitab, 2010] Minitab (2010). "Minitab Inc. home-page".

[Nestor et al., 2008] Nestor, D.; Thiel, S.; Botterweck, G.; Cawley, C.; and Healy, P. (2008). "Applying visualisation techniques in software product lines". In *Proceedings of the 4th ACM symposium on Software visualization (SoftVis'08)*, pages 175–184.

[Nicoara and Alonso, 2005] Nicoara, A. and Alonso, G. (2005). "Dynamic AOP with PROSE". In *Proceedings of the 1st International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05)*, pages 125–138.

[OED, 1989] OED (1989). "Oxford English Dictionary, Online Edition, 1989".

[Oreizy et al., 2008] Oreizy, P.; Medvidovic, N.; and Taylor, R. N. (2008). "Runtime software adaptation: framework, approaches, and styles". In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910.

[Pásztor and Veitch, 2002] Pásztor, A. and Veitch, D. (2002). "PC based precision timing without GPS". *ACM SIGMETRICS Performance Evaluation Review*, 30(1), pp. 1–10.

[Perkins et al., 2009] Perkins, J. H.; Kim, S.; Larsen, S.; Amarasinghe, S.; Bachrach, J.; Carbin, M.; Pacheco, C.; Sherwood, F.; Sidiroglou, S.; Sullivan, G.; Wong, W.-F.; Zibin, Y.; Ernst, M. D.; and Rinard, M. (2009). "Automatically Patching Errors in Deployed Software". In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, pages 87–102.

[Petters et al., 2007] Petters, S. M.; Zadarnowski, P.; and Heiser, G. (2007). "Measurements or Static Analysis or Both?". In *7th International Workshop of Worst-Case Execution Time Analysis*, pages 5–11, Pisa, Italy.

[Poon et al., 2004] Poon, S.-H.; Rockinger, M.; and Tawn, J. (2004). "Extreme Value Dependence in Financial Markets: Diagnostics, Models and Financial Implications". *The Review of Financial Studies*, 17(2), pp. 581–610.

[Pop et al., 2008] Pop, T.; Pop, P.; Eles, P.; Peng, Z.; and Andrei, A. (2008). "Timing analysis of the FlexRay communication protocol". *Real-Time Systems*, 39(1-3), pp. 205–235.

[Popovici et al., 2003] Popovici, A.; Alonso, G.; and Gross, T. R. (2003). "Just-In-Time Aspects: Efficient Dynamic Weaving for Java". *In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109.

[Pretschner et al., 2007] Pretschner, A.; Broy, M.; Kruger, I. H.; and Stauner, T. (2007). "Software Engineering for Automotive Systems: A Roadmap". In *Workshop on the Future of Software Engineering (FOSE '07)*, pages 55–71.

[Ranganath et al., 2003] Ranganath, V. P.; Childs, A.; Greenwald, J.; Dwyer, M. B.; Hatcliff, J.; and Singh, G. (2003). "Cadena: enabling CCM-based application development in Eclipse". In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 20–24.

[Rasche and Polze, 2005] Rasche, A. and Polze, A. (2005). "Dynamic Reconfiguration of Component-based Real-time Software". In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, pages 347–354.

[Redmond and Cahill, 2002] Redmond, B. and Cahill, V. (2002). "Supporting Unanticipated Dynamic Adaptation of Application Behaviour". In *16th European Conference on Object Oriented Programming (ECOOP'02)*, pages 205–230.

[Rosenberg and Schuermann, 2006] Rosenberg, J. V. and Schuermann, T. (2006). "A general approach to integrated risk management with skewed, fat-tailed risks". *Journal of Financial Economics*, 79(3), pp. 569–614.

[Sadjadi et al., 2004] Sadjadi, S. M.; McKinley, P. K.; Cheng, B. H.; and Stirewalt, R. K. (2004). "TRAP/J: Transparent Generation of Adaptable Java Programs". In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, volume 3291, pages 1243–1261.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). "Self-adaptive software: Landscape and research challenges". *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2), pp. 1–42.

[Schellenkens, 2010] Schellenkens, M. P. (2010). "MOQA: unlocking the potential of compositional static average-case analysis". *Journal of Logic and Algebraic Programming*, 79, pp. 61–83.

[Schmid et al., 2008] Schmid, T.; Charbiwala, Z.; Friedman, J.; Cho, Y. H.; and Srivastava, M. B. (2008). "Exploiting manufacturing variations for compensating environment-induced

clock drift in time synchronization". In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '08)*, pages 97–108.

[Schmidt, 2007] Schmidt, H. W. (2007). "Architecture-Based Reasoning About Performability in Component-Based Systems". In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'07)*, pages 130–137.

[Sehlberg et al., 2006] Sehlberg, D.; Ermedahl, A.; Gustafsson, J.; Lisper, B.; and Wiegratz, S. (2006). "Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems". In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, pages 212–219.

[Seinturier et al., 2006] Seinturier, L.; Pessemier, N.; Duchien, L.; and Coupaye, T. (2006). "A Component Model Engineered with Components and Aspects". In *Proceedings of the 9th International Symposium on Component-based Software Engineering (CBSE'06)*, pages 139–153.

[Sharma et al., 2004] Sharma, P. K.; Loyall, J. P.; Heineman, G. T.; Schantz, R. E.; Shapiro, R.; and Duzan, G. (2004). "Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems". In *International Symposium on Distributed Objects and Applications (DOA'04)*.

[Silven and Jyrkkä, 2007] Silven, O. and Jyrkkä, K. (2007). "Observations on power-efficiency trends in mobile communication devices". *EURASIP Journal on Embedded Systems*, 2007(1), pp. 17–27.

[Simon et al., 2006] Simon, D.; Cifuentes, C.; Cleal, D.; Daniels, J.; and White, D. (2006). "Java(TM) on the Bare Metal of Wireless Sensor Devices". In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*, pages 78–88.

[Smith, 2007] Smith, R. B. (2007). "SPOTWorld and the Sun SPOT". In *6th International Symposium on Information Processing in Sensor Networks*, pages 565–566.

[Smith et al., 2005] Smith, R. B.; Cifuentes, C.; and Simon, D. (2005). "Enabling Java^{TM} for small wireless devices with Squawk and SpotWorld". In *OOPSLA Workshop on Bringing Software to Pervasive Computing*.

[Smits et al., 2009] Smits, T.; Vanrompay, Y.; and Berbers, Y. (2009). “Efficient decision making algorithms for adaptive applications”. In *Proceedings of the 3rd International Workshop on Adaptive and Dependable Mobile Ubiquitous Systems (ADAMUS'09)*, pages 13–18.

[Souyris et al., 2005] Souyris, J.; Pavec, E. L.; Himbert, G.; Jégu, V.; and Borios, G. (2005). “Computing the worst case execution time of an avionics program by abstract interpretation”. In *In proceedings of the 5th intl workshop on worst-case execution time analysis (WCET'05)*, pages 21–24.

[Staschulat et al., 2006] Staschulat, J.; Braam, J. C.; Ernst, R.; Rambow, T.; and Busch, R. S. R. (2006). “Cost-Efficient Worst-Case Execution Time Analysis in Industrial Practice”. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, pages 212–219.

[Sun, 2009] Sun (2009). “Sun SPOT Owner's Manual: Red Release 5.0”.

[Sutton et al., 2006] Sutton, P.; Doyle, L. E.; and Nolan, K. E. (2006). “A Reconfigurable Platform for Cognitive Networks”. In *1st International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, pages 1–5, Mykonos, Greece.

[Szyperski and Pfister, 1997] Szyperski, C. A. and Pfister, C. (1997). “Summary of the Workshop on Component Oriented Programming (WCOP'96)”. *Special Issues in Object-Oriented Programming*, pages 127–130.

[Thiel and Hein, 2002] Thiel, S. and Hein, A. (2002). “Modeling and Using Product Line Variability in Automotive Systems”. *IEEE Software*, 19, pp. 66–72.

[Townley et al., 2009] Townley, J.; Manning, J.; and Schellekens, M. (2009). “Sorting Algorithms in MOQA”. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 225, pp. 391–404.

[Upton and Cook, 2004] Upton, G. and Cook, I. (2004). *Oxford Dictionary of Statistics.* Oxford University Press, 2nd edition.

[Vanderperren et al., 2005] Vanderperren, W.; Suvée, D.; Verheecke, B.; Cibrán, M. A.; and Jonckers, V. (2005). “Adaptive programming in JaSco”. In *In Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 75–86.

[Wall et al., 2002] Wall, A.; Larsson, M.; Norström, C.; and Crnkovic, I. (2002). "Using Prediction Enabled Technologies for Embedded Product Line Architectures". In *ICSE workshop on 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, FL, USA.

[Wang et al., 2004] Wang, N.; Gill, C.; Subramonian, V.; and Schmidt, D. C. (2004). "Configuring Real-time Aspects in Component Middleware". In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, pages 1520–1537.

[Wenzel et al., 2005] Wenzel, I.; Kirner, R.; Rieder, B.; and Puschner, P. (2005). "Measurement-based worst-case execution time analysis". In *Proceedings of 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10.

[Westermann and Happe, 2010] Westermann, D. and Happe, J. (2010). "Towards Performance Prediction of Large Enterprise Applications Based on Systematic Measurements". In *Proceedings of the 15th International Workshop on Component-Oriented Programming (WCOP) 2010*, pages 71–78, Karlsruhe, Germany.

[Wilhelm et al., 2008] Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mueller, F.; Puaut, I.; Puschner, P.; Staschulat, J.; and Stenström, P. (2008). "The Determination of Worst-Case Execution Times - Overview of the Methods and Survey of Tools". *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), pp. 1–53.

[Young, 1977] Young, I. T. (1977). "FProof without Prejudice: Use of the Kolmogorov-Smirnov Test for the Analysis of Histograms from Flow Systems and Other Sources". *The Journal of Histochemistry and Cytochemistry*, 25(7), pp. 935–941.

[Zave and Jackson, 1997] Zave, P. and Jackson, M. (1997). "Four dark corners of requirements engineering". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1), pp. 1–30.

[Zhang et al., 2005] Zhang, J.; Cheng, B. H. C.; Yang, Z.; and McKinley, P. K. (2005). "Enabling Safe Dynamic Component-Based Software Adaptation". In *in Architecting Dependable Systems III, Springer Lecture Notes for Computer Science*, pages 194–211. Springer-Verlag.

[Zhang et al., 2009] Zhang, J.; Goldsby, H. J.; and Cheng, B. H. (2009). "Modular Verification of Dynamicaly Adaptive Systems". In *Proceedings of the 8th ACM International Conference on*

Aspect-Oriented Software Development (AOSD'09), pages 161–172. Charlottesville, Virginia, USA.

Glossary

ARMA Auto-Regressive Moving Average, a time series analysis that evaluates the behaviour of data over a particular number of values. See ARIMA model.

ARIMA Auto-Regressive Integrated Moving Average, similar to the ARMA model, except trend information is included (integrated) within the resulting forecasts. Both the ARMA and ARIMA models are traditionally applied to estimate the behaviour of highly-variable time series, such as stock prices and currency markets.

BCET Best-Case Execution Time, the fastest theoretical execution time for the software running within a specified operating environment.

CDF Cumulative Distribution Function, a graph of the probability of a particular statistical distribution being encapsulated by a specified value. The CDF is used within TimePredict to assess how close the fitted distribution matches the underlying measurement data.

Central Tendency The typical, or commonly observed behaviour of a particular process. In the case of timing measurements, the central tendency refers to the clustering of the observed timing measurements about a particular value.

Correlation The correlation between two variables is a measure of the relationship that exists between them, such that a change in one variable is matched by a reciprocal change in the other. See Pearson correlation coefficient.

CPU Central Processor Unit, the engine at the core of a computing system responsible for the execution of individual instructions.

Critical Value The value, for a pre-defined level of confidence, at a particular level of significance, at which a statistical test result may be safely accepted or rejected. Typically the critical value is defined with tables, and compared against the test statistic to determine the outcome of the test.

Dalymount cat One or more specimens of the genus felis catus, likely to make their appearance at crucial periods. Thought to portend a favourable result.

Decile A division of a data set of frequency distribution into tenths, similar to the median dividing a similar set of data into two halves.

ES model The Exponential Smoothing model, a heuristic-based forecasting method that only requires the most recent value from a time series in order to generate an estimate. The ES model forms part of the TimePredict approach, and is applied in situations where limited timing measurements are available, e.g., immediately following adaptations.

Fréchet Distribution — A type of statistical distribution used within the Generalized Extreme Value model. The Fréchet distribution is sometimes known as a Type II Extreme Value distribution, and has an associated shape parameter that can be used to alter the peakedness (kurtosis) of the distribution to better fit a given dataset.

GARCH Model — Generalized Auto-Regressive Conditional Heteroskedasticity model. An auto-regressive time-series forecasting model used traditionally to estimate trends within highly variable environments, such as within currency trading or stock exchange markets.

GEV Model — Generalized Extreme Value, a family of similar statistical models, each having more of the distribution in the tails than the typical Gaussian (Normal) distribution. The GEV model is used within TimePredict to generate an estimate of the worst-case timing behaviour of the software, by fitting one of the available GEV distributions to a frequency distribution of the timing measurement data.

Gumbel Distribution — A type of statistical distribution used within the Generalized Extreme Value model, also referred to as a Type I Extreme Value distribution. Unlike the other Extreme Value distributions, the Gumbel distribution has no associated shape parameter, but instead presents the same skewed bell-shaped curve at all times.

ILP — Integer Linear Programming, an approach towards multi-variable problems that attempts to find the optimum (or worst-case) outcome across all the potential combinations of each variable, often an NP-hard problem.

JAR A Java ARchive file, used to store a collection of Java class files. JAR files can be deployed and executed as a single unit.

Kurtosis A measure of the peakedness of a particular statistical distribution, i.e., how many measurements within a particular dataset cluster around a specific value.

Interdecile Range A measure of the statistical dispersion of a dataset, specifically, the difference between the first and ninth deciles.

Inter-Quartile Range A measure of the statistical dispersion of a dataset, specifically, the difference between the first and third quartiles. The values within the dataset within these quartile bounds for the middle 50% of the available data, i.e., the more common values encountered within the dataset. The mid-point of the inter-quartile range provides the median value of same data.

IQR See Inter-Quartile Range.

Kolmogorov-Smirnov test The Kolmogorov-Smirnov test is a minimum distance test, used to assess the goodness of fit for a particular statistical distribution, based on the sum of the distance of its CDF from the frequency distribution of the dataset.

KS test See Kolmogorov-Smirnov test.

Mann-Whitney Test A non-parametric statistical test used to detect the likely difference between two datasets. Unlike model-based hypothesis tests, the Mann-Whitney test makes no assumptions about the distribution of the underlying data, allowing it to be applied with equal vigor to both normal and various non-Gaussian datasets.

OTA Over-The-Air, a type of wireless software deployment mechanism.

P-value The p-value provides the probability of the same, or a more extreme, test statistic value being generated randomly, when performing a statistical hypothesis test. The p-value is used as an indication of the level of significance of the test result.

Pareto Distribution A type of statistical distribution used within the Generalized Extreme Value model. While not formally part of the Generalized Extreme Value (GEV) family of statistical distributions, it nonetheless provides a useful statistical model in predicting exceedances, or other exponentially-decreasing data series.

Pearson Correlation Coefficient The Pearson product-moment correlation coefficient provides an estimate of the correlation between two variables, within the range -1.0 to +1.0. A correlation coefficient in excess of 0.7 (either positive or negative) provides a good indicator for a strong relationship between the two variables under test. Any coefficient typically within the range -0.3 to +0.3 is considered to show no correlation.

PDF Probability Density Function, a measure of the probability of generating a particular value from a given statistical distribution. Integrals under the range of the PDF are used to define the probability of a randomly selected value falling between two specified bounds. Within the Gaussian distribution, the PDF assumes a bell-shaped curve, whereas other statistical distributions, e.g., the GEV models, may exhibit more skewed or asymmetric shapes.

Quartile Dividing an ordered data set into quarters, so that each subset contains precisely one quarter of the total elements of the set. The quartile is the element that forms the boundary value between these quarters, similar to the median dividing the data set into two equal halves. Quartiles are typically labeled Q_1 to Q_3, with Q_0 being the minimum and Q_4 the maximum.

T-value The test statistic produced by either the independent or paired T-test, two statistical hypothesis tests that use Student's T-distribution to evaluate the likely difference in means between two datasets. See T-test.

T-test The T-test is a statistical hypothesis test to evaluate whether a statistically significant difference exists between the means of two datasets. The T-test can be either independent, where no relationship exists between the two tested datasets, or paired, where the same process is tested twice, e.g., before-and-after tests. The strengths of the T-test are its use of a statistical model to assess the likely differences in the means, however a number of underlying assumptions must be met concerning similar sample sizes, variances and normality within the tested data.

Test Statistic The value produced by a statistical test when evaluating one or more datasets. The test statistic must be compared to the critical value to assess the outcome of the test. Within statistical hypothesis tests, any test statistic greater than the critical value typically results in rejecting the null hypothesis. See critical value.

Weibull Distribution A type of statistical distribution used within the Generalized Extreme Value model. The Weibull distribution is often referred to as a Type III Extreme Values distribution. The Weibull distribution can take on both an exponentially decreasing curved shape, or a skewed bell-curve shape, depending on the value of its shape parameter α.

WCET Worst-Case Execution Time, the slowest theoretical execution time for the software running within a specified operating environment.