Resource modeling of software components in the OSAS framework

by

Ashenafi Tesfay Gebreweld

A thesis submitted in partial fulfillment for the Master’s Degree

in the

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Graduation supervisor
Prof. Dr. Johan J. Lukkien
Graduation tutor
Dr. Ir. Richard P.H.F.M. Verhoeven

July 2010
Abstract

Wireless sensor networks (WSN) are resource constrained systems. Therefore, determining resource requirements of WSN applications becomes important. This thesis presents the design and implementation of Execution time (ET) and memory models for OSAS (Open Service Architecture for Sensors) software components. The ET analysis tool, written in C++ (5000 lines-of-code), gives the best-case and worst-case execution times of OSAS execution units. It also provides the ET breakdown in terms of program constructs such as while loops and conditional statements. The memory analysis tool, written in C++ (600 lines-of-code), can predict the memory usage of OSAS software components with only 3% error.
Acknowledgements

I thank my graduation supervisor Prof. Dr. Johan J. Lukkien, and my graduation tutor Dr. Ir. Richard P.H.F.M. Verhoeven for providing me with excellent supervision throughout the thesis project. I also thank Dr. Ir. Sander Stuijk for being in my assessment committee. I also thank Ir. R.P. (Remi) Bosman for providing help in relation to the OSAS. Last but not least, I thank the remaining SAN members for building a good working environment from which I benefited.
# Contents

## Abstract

iii

## Acknowledgements

v

## List of Figures

xi

## List of Tables

xiii

## Abbreviations

xv

1 Introduction 1

1.1 Introduction to wireless sensor networks (WSN) 1

1.1.1 WSNs and resources 1

1.2 Introduction to OSAS 2

1.2.1 OSAS software components 5

1.2.1.1 Services 5

1.2.1.2 Subscriptions 7

1.2.1.3 CBAs 8

1.2.2 OSAS execution units 8

1.2.3 Deployment 9

1.3 The problem statement 10

1.3.1 Title 10

1.3.2 Graduation assignment and plan 10

1.3.3 Description of work 11

1.4 Selected resources 12

1.5 Organization of the thesis report 12

2 Measurement-based Execution Time (ET) Model 13

2.1 Introduction 13

2.2 State of the art in ET analysis 13

2.2.1 ET analysis approaches 16

2.2.1.1 Static ET analysis 16

2.2.1.2 Measurement-based ET analysis 17

2.2.1.3 Design decision: Static or measurement-based ET analysis 17

2.2.2 Interpreted code on top of an embedded microprocessor 17
2.2.3 Commercial ET analysis tools ........................................ 18
  2.2.3.1 aiT ......................................................... 18
  2.2.3.2 Bound-T .................................................. 19
  2.2.3.3 RapiTime .................................................. 19

2.3 Motivations for developing an ET model for OSAS software components 19

2.4 OSAS program analysis .................................................. 20
  2.4.1 OSAS virtual machine instruction set ............................... 20
  2.4.2 OSAS program constructs .......................................... 23
    2.4.2.1 Basic blocks ............................................ 25
    2.4.2.2 Conditional blocks ...................................... 25
    2.4.2.3 While blocks ............................................ 25
    2.4.2.4 If-then blocks ........................................... 25
    2.4.2.5 Example program .......................................... 26
    2.4.2.6 OSAS bytecode block composition ........................... 26
  2.4.3 Block identification and delineation .............................. 26
    2.4.3.1 Definitions .............................................. 26
    2.4.3.2 Notations ............................................... 26
    2.4.3.3 Order for block identification ............................ 28
    2.4.3.4 An algorithm for identifying while blocks in bytecode ... 28
    2.4.3.5 An algorithm for identifying conditional blocks in bytecode 30
    2.4.3.6 An algorithm for identifying if-then blocks in bytecode ... 37
    2.4.3.7 An algorithm for identifying basic blocks in bytecode ... 37
    2.4.3.8 A note on the efficiency of the algorithms ................ 37
    2.4.3.9 Delineating while blocks ................................ 37
    2.4.3.10 Delineating conditional blocks ........................... 38
    2.4.3.11 Delineating if-then blocks ................................ 40
    2.4.3.12 Delineating basic blocks ................................ 41
    2.4.3.13 A note on the tests of while, conditional, and if-then blocks 41
    2.4.3.14 Summary: Block identification and delineation ............ 43
  2.4.4 Determining the ET of an OSAS program ............................ 43
    2.4.4.1 Determining the WCET and BCET of a bytecode: Exam- ... 45
    2.4.4.2 The lookup data .......................................... 52
    2.4.4.3 The execution time of the 44 Bytecode instructions ...... 53
    2.4.4.4 A child table for CALL[id] ................................ 55
    2.4.4.5 A table of execution time of administrative tasks ...... 56
    2.4.4.6 LED toggling overhead ................................... 58

2.5 Platform: TelosB running MANTIS Operating System .................. 45
  2.5.1 MSP430 ....................................................... 45
  2.5.2 Mantis Operating System ........................................ 49
  2.5.3 The OSAS thread ............................................... 49
  2.5.4 Timer interrupt ............................................... 50

2.6 Measurements .................................................................. 51
  2.6.1 Measurement setup ............................................... 51
    2.6.1.1 Interpreter instrumentation ............................... 51
  2.6.2 The lookup data ............................................... 52
    2.6.2.1 The execution time of the 44 Bytecode instructions ...... 53
    2.6.2.2 A child table for CALL[id] ................................ 55
    2.6.2.3 A table of execution time of administrative tasks ...... 56
    2.6.2.4 LED toggling overhead ................................... 58

2.7 ET analysis tool .......................................................... 58
  2.7.1 Using the ET analysis tool: Example ............................... 58
3 Memory model

3.1 Introduction

3.2 Linked-lists

3.2.1 Internal fragmentation overhead

3.2.2 External fragmentation overhead

3.3 Memory layout of OSAS

3.4 The memory models

3.5 Measurements

3.5.1 Measurement setup

3.6 Memory analysis tool

3.6.1 Using the memory analysis tool: Example

3.7 Memory model evaluation

4 Conclusions and further work

4.1 Summary and Conclusions

4.1.1 The ET analysis tool

4.1.2 The memory analysis tool

4.1.3 Revisiting the problem statement

4.1.3.1 static and dynamic parts of the resource models

4.1.3.2 Measuring and specifying the dependence of resource use on run-time parameters

4.1.3.3 Model integration and evaluation

4.1.3.4 What to Use the models for

4.2 Further work

4.2.1 Other resources

4.2.2 Taking CBAs into consideration

4.2.3 Determining the #iterations of while loops

4.2.4 Extending the ET analysis tool with nesting
List of Figures

1.1 Thesis title ......................................................... 2
1.2 OSAS application development process without a resource model .... 3
1.3 OSAS application development process helped by a resource model ... 4
1.4 The general structure of an OSAS service .......................... 5
1.5 A service that reads and sends temperature readings ................. 6
1.6 A service with all parts ............................................ 6
1.7 Subscriptions specify the activation period of the EG chain of a service. 7
1.8 Handlers are mostly call-back functions ............................ 7
1.9 Most common relationships between service parts .................... 8
1.10 A deployment example ............................................ 9
1.11 Uploading the program of Listing 1.2 ............................. 10
2.1 Problem domains of WCET analysis (Puschner et al. [1]) ............ 14
2.2 [2] Basic notions concerning timing analysis of systems. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal and maximal observed execution times, respectively. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best- and worst-case execution times, respectively, abbreviated BCET and WCET ....................... 15
2.3 Interpreted vs. compiled systems: OSAS is an interpreted system .... 21
2.4 OSAS opcodes: Encoding .......................................... 23
2.5 RJFs and JUMPs ..................................................... 24
2.6 Example program containing the four types of blocks ................. 27
2.7 OSAS bytecode block composition .................................. 28
2.8 A pictorial depiction of the algorithm for identifying while blocks (Section 2.4.3.4) ................................................. 29
2.9 Identifying a while block visually ................................... 30
2.10 Identifying a while block using the algorithm ....................... 30
2.11 Identifying two conditional blocks visually ......................... 33
2.12 Test for condition 3 on the five sequences .......................... 36
2.13 Delineating while blocks .......................................... 38
2.14 If-then(elif-then)*-else constructs: General structure .............. 39
2.15 Delineating conditional blocks: Example .......................... 41
2.16 Delineating conditional blocks: Determining WCET and BCET ...... 42
2.17 Delineating if-then blocks: Example ................................ 43
2.18 Block identification and delineation: Order .......................... 44
2.19 Breakdown of ET of execution units ................................ 45
2.20 Determining the ET of a bytecode: source and bytecode ............ 46
List of Figures

2.21 Determining the ET of a bytecode: WCET ........................................ 47
2.22 Determining the ET of a bytecode: BCET ........................................ 48
2.23 The MANTIS Operating System ..................................................... 49
2.24 The OSAS thread ................................................................. 50
2.25 Measurement setup .............................................................. 51
2.26 Measurement: Example .......................................................... 52
2.27 Time line of ET of execution units .............................................. 57
2.28 ET model implementation ......................................................... 58
2.29 ET analysis tool ................................................................. 59
2.30 ET model evaluation .............................................................. 61

3.1 Linked lists ................................................................. 64
3.2 The hierarchical structure of the MMU .......................................... 64
3.3 Internal and External fragmentation overheads: Example ................. 66
3.4 OSAS memory layout .............................................................. 67
3.5 The layout of the scenario .......................................................... 71
3.6 The Loader and Simulator before the base scenario is installed .......... 72
3.7 The Loader and Simulator when base scenario is running ................. 73
3.8 Memory map after the base scenario is installed .............................. 73
3.9 The Loader and Simulator when the main scenario is running .......... 75
3.10 Memory model evaluation .......................................................... 78
3.11 Model 3 → Model 1 → Model 2 → Model 4 in increasing order of over/under estimation .......................................................... 79
3.12 Model 3 provides the closest estimate (3% error). ........................ 79
List of Tables

2.1 WCET analysis research: Contributions of some Institutions . . . . . . . 15
2.2 A brief description of the OSAS instruction set[3] . . . . . . . . . . . . . 22
2.3 Step 1: JUMPs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.4 Step 1: RJFs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.5 Group 1 sequences . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.6 A partial view of Group 2 sequences . . . . . . . . . . . . . . . . . . . . . 34
2.7 The execution time of the 44 Bytecode instructions . . . . . . . . . . . . 54
2.8 A child table for CALL[id] . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
2.9 A table of execution time of administrative tasks; precision=1\mu s . . . . 56

3.1 OSAS header sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.2 The memory models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSAS</td>
<td>Open Service Architecture for Sensors</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
<tr>
<td>EG</td>
<td>Event Generator</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>WaSCoL</td>
<td>Wasp Service Composition Language</td>
</tr>
<tr>
<td>BC</td>
<td>Byte Code</td>
</tr>
<tr>
<td>WBC</td>
<td>WaSCoL Byte Code</td>
</tr>
<tr>
<td>ET</td>
<td>Execution Time</td>
</tr>
<tr>
<td>WASP</td>
<td>Wirelessly Accessible Sensor Populations</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Controller Unit</td>
</tr>
<tr>
<td>MANTIS</td>
<td>Multimodal system for Networks of In-situ wireless Sensors</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>BCET</td>
<td>Best-Case Execution Time</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>CBA</td>
<td>Content Based Address</td>
</tr>
<tr>
<td>B</td>
<td>Byte</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction to wireless sensor networks (WSN)

WSNs are formed by a group of small-capacity, (wirelessly) communicating nodes. The wireless nodes have sensing, computation, communication, and actuation functionalities. Some of their applications are in the areas of agriculture, health care, and environment monitoring.

1.1.1 WSNs and resources

Wireless sensor networks are resource constrained systems. Some of these constrained resources are energy, processing power, memory, and communication bandwidth.

The potential applications for WSNs require that the wireless nodes be as small in size as possible. This puts a limit on the size of a battery that we can put with the wireless nodes. Therefore we have to use the battery efficiently such that the life of the wireless sensor network is maximized. To maximize network lifetime, we have to minimize the amount of communication. This in turn means putting a limit on message traffic. To ensure that WSN applications function correctly and with the required quality of service, it becomes important to build models for the resources so that resource requirements can be determined and optimized at development time. In this thesis, we design and implement resource models for OSAS (Open Service Architecture for Sensors) software components.
1.2 Introduction to OSAS

This thesis is about resource modeling of software components in the OSAS framework (See Figure 1.1). Resource modeling of software components is the activity of designing/specifying a model which estimates/bounds the resource usage of software components when they are uploaded to a target platform (See Section 1.3, “The problem statement” for the complete problem statement).

Figure 1.1: Thesis title

OSAS is an integrated environment for programming wireless sensor networks[3, 4]. It was developed at the SAN group, TU/e. OSAS allows sensors to be reprogrammed over the network by translating a single program into a set of configuration messages and bytecode.

The OSAS toolchain consists of a simulator, a compiler, a loader, and an interpreter. The simulator is used to test for correctness of application functionality.

This thesis provides an initial work in the direction of adding one other component: Resource model.

Figure 1.2 shows the application development process using the OSAS toolchain. We see from the figure that applications are uploaded without checking their resource usages with respect to what is available at the target node. On the other hand, Figure 1.3
shows the application development process with a resource model used to check the resource usage of the OSAS software components before they are uploaded. The resource model is an important part of the application development process. For soft real-time applications, the model can be made to provide resource usage estimates. For hard real-time applications, the model should be made to provide bounds rather than estimates.

---

**Figure 1.2**: OSAS application development process without a resource model
Figure 1.3: OSAS application development process helped by a resource model
1.2.1 OSAS software components

The OSAS software components are services, subscriptions, and content based addresses (CBA).

1.2.1.1 Services

A service is an installable component which is responsible for providing a certain functionality on nodes. Examples: Temperature service, Acceleration service

The general structure of an OSAS service is shown in Figure 1.4. The main parts of an OSAS service are: State, functions, event-generators (EG), and handlers. The EGs of a service are also collectively known as an EG chain. Figure 1.5 shows a service which has only a single EG to read and send temperature values. Figure 1.6 shows a bigger services with all parts.

![Figure 1.4: The general structure of an OSAS service](image)

The service state holds global variables. OSAS service functions are like any other function in other programming environments/languages. They can be called by EGs, handlers, or other functions. An event generator chain fires at periods specified by subscriptions (See Figure 1.7). Handlers are mostly used as call-back functions (See Figure 1.8).
service TempService($Handler)
  on event measure when True do
    SendToSubscribers($Handler, NodeID(), Temp())

Figure 1.5: A service that reads and sends temperature readings

---

service Service1($Handler, $a, $b)
  define
    arr1[10] := []
    sum := 0
    c := 0

  function init() do
    j := 0;
    while j < 10 do
      arr1[j] := j
    od

  function average() do
    i := 0;
    while i < 10 do
      sum := sum + arr1[i];
      i := i + 1
    od;
    return (sum / 10)

  on event e1 when $a == $b do
    init();
    a := average();
    SendToSubscribers($Handler, a)

  on event e2 when True do
    init();
    SendToSubscribers($Handler, *arr1)

  action action1() do
    init();
    sum := 0

  action action2() do
    c := average()

Figure 1.6: A service with all parts
1.2.1.2 Subscriptions

Nodes use services on other nodes by subscribing to them. A service installed on a node does not do anything useful by itself unless some other nodes subscribe to it. Listing 1.1 shows an example subscription to the service given by Figure 1.5. Through the subscriptions, we provide values to subscription variables of the target services and we set period for the activation of EG chains. The fields “deadline”, “send”, and “exec” are not implemented yet. When implemented, they will enable quality-of-service decisions in the wireless sensor network.

```
subscription TempServiceSubscription
to TempService($Handler=print)
with (period=20 ms, deadline=1s, send="Normal", exec="Normal")
```

Listing 1.1: A subscription to the service in Figure 1.5
1.2.1.3 CBAs

CBAs are addresses where the nodes are not identified with preset IDs but with the functionality (content) they provide (have). An example of a CBA is \([\text{Network}] \ast [\text{Temp()} > 40]\). This CBA identifies nodes which are at any hop count (starting from 0) with their Temp() reading greater than 40. Another example of a CBA is \([\text{Network}[2]]\). This CBA identifies all nodes within the distance of two hops. CBAs enable data-centric programming.

1.2.2 OSAS execution units

The OSAS execution units are CBAs, functions, EGs, and handlers. CBAs are evaluated at message arrivals. Functions execute when they are called. EGs execute periodically as specified by subscriptions. Handlers normally execute at a subscriber node when notification messages are processed. Note that subscriptions are not execution units.
Subscriptions only enable linking between The EGs of one service and the handlers of another service.

### 1.2.3 Deployment

Deployment is the act of configuring a wireless sensor network with an application. Figure 1.10 shows a deployment example. Listing 1.2 shows a complete OSAS program. After that program is compiled, it is deployed using the Loader (See Figure 1.11).

**Deploying a subscription**

```
service TempService($Handler)
    on event measure when True do
        SendToSubscribers($Handler, NodeID(), Temp())

subscription TempServiceSubscription to TempService($Handler=print)
    with (period=20 ms, deadline=is, send="Normal", exec="Normal")

for [Network[*]|NodeType()=='TemperatureNode']
    install TempService

for [Network[*]|HasService(TempService)]
    install TempServiceSubscription on [Network[*]|NodeType()=='TemperatureNode']
```

---

**Figure 1.10: A deployment example**

---

**Listing 1.2: A complete OSAS program**
1.3 The problem statement

1.3.1 Title

Resource modeling of Software Components in the OSAS framework

1.3.2 Graduation assignment and plan

The topic of the project is an investigation into predictive resource models for software components. With resources we mean: memory, processing power, and communication bandwidth. Computed from this, when considering a particular hardware are time (throughput, latency and jitter), energy and utilization. (E.g., given a certain bandwidth requirement and the bandwidth of the real system a utilization can be computed.)

\footnote{As written by Prof. Johan J. Lukkien.}
We seek answers to the following questions:

- What are static (derived during the compilation/link procedure) and dynamic parts (depending on inputs during execution) of the resource model?

- How can we measure the dependence of resource use on run-time parameters?

- How can we specify the dependence of resource use on run-time parameters? Which format is useful to describe the resource model?

- How to combine the resource model with the component in some sort of container? How to load and certify?

- How to use such models for
  - an acceptance test (computing the effect of adding a component to an existing composition);
  - a predictive feedback (e.g. the effects on energy of certain settings);
  - continuous monitoring.

When time permits we also want to make the model adaptive, i.e., learning from the dynamics the software component it is involved in.

The research focuses on the component model used in OSAS and this is also the context for the interpretation and application of the results. However, we want to relate this to component-based software in general.

Outcome of the project are answers to the above questions, supported by experiments with at least two different setups: one with a single node performing different logging applications and one with up to ten nodes with multi-hop requirements.

In addition, a research paper is written as part of the project.

1.3.3 Description of work

The initial phases of the work include a) getting to know component-based software (what is a component model, what are elements of an implementation); b) understanding resource descriptions; c) learning about wireless sensor networks and OSAS, and relating
the OSAS component model to general component models. Literature will be selected for that purpose.

Subsequently, the above questions will determine the agenda. We aim at selecting a limited number of resources to model and perform the entire research and test cycle. This model can further be extended. During the course of the project we take further decisions to control the amount of work involved.

### 1.4 Selected resources

We selected CPU (execution time) and memory to be the resources that we will model in this thesis. Execution time model is important for understanding the timing behavior of WSN applications. For OSAS, the execution time model is also important for setting subscription periods (See Section 2.3 for motivations of developing an execution time model for OSAS). Since memory is a constrained resource in wireless nodes, the memory model we developed will help the OSAS application developers tune their applications so that the applications fulfill memory requirements. We indicate other resources that are important to model for WSNs in Section 4.2.

### 1.5 Organization of the thesis report

In chapter 2, “Measurement-based Execution Time (ET) Model”, we present the design and implementation of the execution time model for OSAS software components. In that chapter, a timing analysis tool is developed and evaluated. In Chapter 3, “Memory model”, we present the memory model for OSAS software components. In that chapter, a memory analysis tool is developed and evaluated. Chapter 4 summarizes and concludes the thesis report. We also revisit the research questions in the problem statement. Furthermore, we indicate some further work.
Chapter 2

Measurement-based Execution Time (ET) Model

2.1 Introduction

The Execution Time (ET) of a task is the amount of time a platform needs to execute the task. Among the motivations for developing an ET model for OSAS software components are to set subscription periods, to determine the ET contributions of different parts of an OSAS program, and to determine the quality of implementation. The shortest ET is called the best-case execution time (BCET); the longest ET is called the worst-case execution time (WCET).

In this chapter we design and implement an ET model for OSAS software components. We first look into the state of the art in ET analysis. We then motivate the need for developing an OSAS ET model. The rest of the chapter presents the design and implementation of the ET model. The final sections present the ET analysis tool and the evaluation of the ET model.

2.2 State of the art in ET analysis

Peter Puschner and Alan Burns reviewed the research field of WCET analysis in their paper “A review of Worst-case Execution Time Analysis” which appeared as an editorial\(^1\)

\(^1\)Peter Puschner and Alan Burns are key members of the WCET analysis community\(^2\).
in the International Journal of Time-Critical Computing Systems in 2000 (Puschner et al. [1]). In this paper it is indicated that the first contributions in WCET analysis appeared about 20 years ago (Kligerman and Stoyenko, 1986 [5]; Mok et al., 1989 [6]; Puschner and Koza, 1989 [7]; Shaw, 1989 [8]). The paper presents the problem domains of WCET analysis (See Figure 2.1). According to these problem domains, two different approaches of ET analysis are identified: static and measurement-based. We will discuss these two approaches in Section 2.2.1.

In Figure 2.1, the characterization of execution paths takes place on the source-language level. The source code and the path information are translated to the machine language representation to analyze its WCET. WCET analysis depends on a good translation of path information. The hardware-level execution-time analysis computes a WCET bound by investigating the execution times of the possible program paths.

Table 2.1 shows the contributions of some institutions to the WCET analysis research.

The ET research is mostly driven by the need to guarantee the proper behavior of hard real-time systems [12, 13]. Figure 2.2, taken from [2], shows the timing-related properties of real-time tasks. Depending on the state of a system, the execution time of tasks varies. The set of all execution times is shown as the upper curve. The shortest execution time is called the best-case execution time (BCET); the longest time is called the worst-case
Table 2.1: WCET analysis research: Contributions of some Institutions

<table>
<thead>
<tr>
<th>Institution</th>
<th>Contribution</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of Toronto</td>
<td>Real-Time Euclid</td>
<td>Kligerman et al. [5]</td>
</tr>
<tr>
<td>University of Washington</td>
<td>Timing Schema</td>
<td>Shaw [8]</td>
</tr>
<tr>
<td>University of Texas at Austin</td>
<td>Timetool</td>
<td>Mok et. al [6]</td>
</tr>
<tr>
<td>Vienna University of Technology (TU Vienna)</td>
<td>TuBound</td>
<td>Prantl et al. [9, 10]</td>
</tr>
</tbody>
</table>

execution time (WCET). To determine the exact values of the worst-case and best-case execution times, all possible executions have to be explored. This is in most cases practically impossible.

The commonly used method of execution time estimation is to measure the end-to-end execution time of a task using test-cases. From this, the minimal observed and maximal observed execution times are determined. This method will, in general, overestimate the BCET and underestimate the WCET [2]. Detailed measurements of the parts of a task can also be done to get a better estimate of the ET. Any measurement-based ET analysis technique provides only an estimation of the BCET/WCET, but it doesn’t provide a bound.
ET bounds can be computed only through methods which consider all possible executions of a task (shortest/longest path analysis). These methods are very complex due to increasingly complex modern processor architectures. Therefore, these methods use abstraction of the task to make timing analysis feasible. The amount of abstraction done depends the required accuracy. Only hard real-time activities need a safe WCET analysis. For the soft real-time activities a probabilistic timing analysis is sufficient to guarantee aspects like quality of service[2].

Modern processor architectures contain many features meant to increase performance. Examples of these features are caches, pipelines, and branch prediction. These features complicate timing analysis as the ET of an instruction is not constant but depends on the state of the processor. In case the timing of instructions does not depend on the processor-state (that is, the processor doesn’t have caches and pipelines), the ET of each instruction can be statically determined.

The two main tasks in execution time analysis are the development of the program control flow (control flow analysis) and the mapping of the program control flow to the processor (processor behavior analysis) [14].

**2.2.1 ET analysis approaches**

Currently, there are two main methods for ET analysis: Static analysis and measurement-based. Measurement-based ET analysis is mostly used with soft real-time applications where an estimate, and not a safe bound, of ET is required. Static analysis is mostly used with hard real-time applications as it provides a safe bound on ET. We now discuss these two ET analysis methods.

**2.2.1.1 Static ET analysis**

Static ET analysis uses program analysis and explicit hardware models to derive a BCET/WCET bounds. (Puschner et al. [15])Static analysis tools analyze the code of the tasks for possible execution paths and then model the timing of the code on the target hardware. While the latter step of the analysis can be fully automated the first step usually has to be supported by the programmer. The programmer annotates the task code with information about the possible execution paths. Some of the contributions
for static analysis are: Kligerman et al. [5], Puschner et. al [7], Ko et al. [16], Li et al. [17], and Puschner et. al [15].

The main challenge for static analysis is the high complexity of today’s processors and the short hardware life cycles[14]. Short hardware life cycles mean frequent update of the tools/methods. This is a big challenge as developing a static analysis tool for a certain processor architecture takes a considerable amount of time.

2.2.1.2 Measurement-based ET analysis

Measurement-based ET analysis techniques use execution-time measurements to construct a hardware model (Kirner et al. [18]). Kirner et al. [19] presents a WCET analysis framework using runtime measurements together with static program analysis. It uses automatic generation of test data to derive the instruction timing of code sequences. Program paths are decomposed into subpaths to make execution-time analysis based on runtime measurements feasible. Other contributions for measurement-based WCET analysis are: Wenzel et al. [20], Deverge et al. [21], Schaefer et al. [22], Petters et al. [23].

2.2.1.3 Design decision: Static or measurement-based ET analysis

The platform that we used for ET analysis in this thesis (See Section 2.5, “Platform”) is such that the processor (MSP430) doesn’t have caches, pipelines, and branch prediction[24]. This is typically the case with the CPUs of wireless sensor nodes (Example: TelosB and BSN nodes). Also, WSN (OSAS) applications are mostly soft real-time applications. Due to these reasons, we use the measurement-based method in this thesis. Namely, we use static program analysis to identify and delineate the program constructs, and we use a lookup data to calculate the ET of the basic blocks.

2.2.2 Interpreted code on top of an embedded microprocessor

OSAS is an interpreted system. Interpreted systems have the advantage of reduced code size and the disadvantage of order-of-magnitude larger execution time. Typical interpreted program instructions have speeds 5-20 times slower than native code and
are up to 2 times smaller [25, 26]. Interpreted code for the TriMedia VLIW processor is 8 times slower than native code and is 5 times smaller[27].

Bate, Bernat, and Puschner [28] presented a framework for performing portable WCET in Java Virtual Machine architecture that it is based on the separation of the WCET analysis in two parts, a machine independent part and a machine dependent part. The machine independent part is performed off-line. Other similar contributions are [29–33].

In [28], in order for the code to be analyzable it has to include annotations on the worst case behavior of its constructs (i.e. maximum loop bounds). Similarly the platform specific details are analyzed offline to derive a virtual machine timing model. The virtual machine timing model consists of definitions of how long each Java bytecode takes to execute, in the worst case, on the given platform.

Hu et al.[34] provides a static timing analysis environment for the development of real-time applications on the Java architecture. The major contributions include introducing a novel Extensible Annotations Class (XAC) format to capture portable annotations from the source level, presenting how to integrate XACs with portable Worst-Case Execution Time (WCET) analysis, describing how to obtain real-time thread parameters from Real-Time Java’s specifications, and demonstrating how static timing analysis using the Java architecture can be carried out from portable code.

2.2.3 Commercial ET analysis tools

We now present a summary of selected commercial ET analysis tools.

2.2.3.1 aiT

aiT[2, 35] WCET Analyzers compute tight bounds for the WCET of tasks in real-time systems statically. They operate on binary executables. They use the IPET (Implicit Path Enumeration Technique) to calculate bounds. The supported hardware platforms are: Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85 (prototype), and Infineon TriCore 1.3.
2.2.3.2 Bound-T

Bound-T\cite{2, 36} is a tool for static WCET analysis from binary executable code. It analyses compiled and linked executables to find the WCET, flow graphs, call graphs, and stack usage. The supported hardware platforms are: Intel-8051 series (MCS-51), Analog Devices ADSP-21020, ATMEL ERC32 (SPARC V7), Renesas H8/300, ARM7 (prototype) and ATMEL AVR and ATmega (prototypes).

2.2.3.3 RapiTime

RapiTime\cite{2, 37} is a measurement-based tool which derives timing information of how long a basic block takes to run from measurements. Measurement results are combined according to the structure of the program to determine an estimate for the longest path through the program.

The RapiTime tool is structure-based and works on a tree representation of the program. The structure is derived from either the source code or from the direct analysis of executables. The timing of individual blocks is derived from extensive measurements extracted from the real system. RapiTime not only computes the maximum of the measured times, but whole probability distributions\cite{38, 39}. The WCET estimates are computed using an algebra of probability distributions.

The methods that RapiTime employs are significantly similar to the methods we followed for the OSAS software components ET model we present in this chapter.

2.3 Motivations for developing an ET model for OSAS software components

Some of the motivation for developing an ET model for OSAS software components are to set subscription periods, to determine the ET contributions of different parts of an OSAS program, and to determine the quality of implementation.

The subscription period is decided by the ET of the EG chain, and the subscription period should be larger than the ET of the EG chain. Incorrectly set subscription periods lead to schedule drift (utilization > 1), and thereby incorrect behavior of the
application. We use the ET analysis tool that we built (See Section 3.6, “ET analysis tool) to determine the ET of the EG chain of a service and set subscriptions periods accordingly.

The knowledge of the ET contributions of different parts of a program can help tune the application for performance. The ET analysis tool that we built (See Section 3.6, “ET analysis tool”) gives the ET breakup of a program among its building blocks.

As increasingly complex tasks are being deployed on sensor networks, it becomes important to model the execution behavior of tasks. Schedulability analysis is needed for deploying hard/soft real-time applications. For schedulability analysis, we need to first determine the ET of each of the tasks under consideration, for which the ET analysis tool that we built can be used. This thesis doesn’t treat schedulability analysis any further.

2.4 OSAS program analysis

OSAS is an interpreted system as opposed to a compiled one (See Figure 2.3). A virtual machine (interpreter) runs on the nodes to execute application bytecode.

2.4.1 OSAS virtual machine instruction set

There are a total of 44 OSAS virtual machine opcodes. Figure 2.4 gives the encoding of the virtual machine opcodes. Table 2.2 gives a brief description of the instruction set[3].

We are particularly interested in RJF and JUMP instructions because they signal branching in the program control flow.

RJF (Relative Jump on False) is an instruction of the form RJF \( x \) where \( x \) is the number of bytes to skip forward if the test just above the RJF results to false (See Figure 2.5). RJFs occur both in conditional (if-then-(elif-then)*-else) and while loop constructs.

JUMP (absolute jump) is an instruction of the form JUMP \( y \) where \( y \) is the absolute byte index to jump to (See Figure 2.5). In the case of conditional constructs, \( y \) is strictly

Figure 2.3: Interpreted vs. compiled systems: OSAS is an interpreted system.
## Chapter 2. Measurement-based Execution Time (ET) Model

<table>
<thead>
<tr>
<th>Type</th>
<th>Mnemonic</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reading memory</strong></td>
<td>PUSHC</td>
<td>const</td>
<td>Pushes a constant onto stack</td>
</tr>
<tr>
<td></td>
<td>PUSHL</td>
<td>id</td>
<td>Pushes local var id onto the stack</td>
</tr>
<tr>
<td></td>
<td>PUSHA</td>
<td>id</td>
<td>Pushes Mcontext[id] onto the stack</td>
</tr>
<tr>
<td></td>
<td>PUSHV</td>
<td>id</td>
<td>Pushes Scontext[id] onto the stack</td>
</tr>
<tr>
<td></td>
<td>PUSHG</td>
<td>id</td>
<td>Pushes Gcontext[id] onto the stack</td>
</tr>
<tr>
<td></td>
<td>POP</td>
<td></td>
<td>Pops a value from the stack</td>
</tr>
<tr>
<td></td>
<td>PUSHS</td>
<td>id</td>
<td>Pops index and offset and pushes the dereference of offset+index</td>
</tr>
<tr>
<td><strong>Writing memory</strong></td>
<td>STOREL</td>
<td>id</td>
<td>Stores the top stack element in local variable id</td>
</tr>
<tr>
<td></td>
<td>STOREG</td>
<td>id</td>
<td>Stores the top stack element in Gcontext variable id</td>
</tr>
<tr>
<td></td>
<td>STOREV</td>
<td>id</td>
<td>Stores the top stack element in Scontext variable id</td>
</tr>
<tr>
<td></td>
<td>STOREA</td>
<td>id</td>
<td>Stores the top stack element in Mcontext variable id</td>
</tr>
<tr>
<td></td>
<td>STORES</td>
<td></td>
<td>Pops value, index and offset and stores the value in the address offset+index</td>
</tr>
<tr>
<td><strong>Special</strong></td>
<td>CALL</td>
<td>id</td>
<td>Performs system call id (arguments are taken from the stack)</td>
</tr>
<tr>
<td></td>
<td>NTFY</td>
<td></td>
<td>Sends a message containing the stack contents to all subscribers in Scontext</td>
</tr>
<tr>
<td></td>
<td>FLOOD</td>
<td></td>
<td>Floods a message containing the stack contents into the network</td>
</tr>
<tr>
<td></td>
<td>EXTEND</td>
<td></td>
<td>Enables the extension of argument bits of an opcode</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD</td>
<td></td>
<td>$S, x, y \rightarrow S, x + y$</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td></td>
<td>$S, x, y \rightarrow S, x - y$</td>
</tr>
<tr>
<td></td>
<td>MUL</td>
<td></td>
<td>$S, x, y \rightarrow S, x \times y$</td>
</tr>
<tr>
<td></td>
<td>DIV</td>
<td></td>
<td>$S, x, y \rightarrow S, x/y$</td>
</tr>
<tr>
<td></td>
<td>MOD</td>
<td></td>
<td>$S, x, y \rightarrow S, x % y$</td>
</tr>
<tr>
<td><strong>Comparison</strong></td>
<td>LESS</td>
<td></td>
<td>$S, x, y \rightarrow S, x &lt; y$</td>
</tr>
<tr>
<td></td>
<td>MORE</td>
<td></td>
<td>$S, x, y \rightarrow S, x &gt; y$</td>
</tr>
<tr>
<td></td>
<td>GREAT</td>
<td></td>
<td>$S, x, y \rightarrow S, x \geq y$</td>
</tr>
<tr>
<td></td>
<td>LEQ</td>
<td></td>
<td>$S, x, y \rightarrow S, x \leq y$</td>
</tr>
<tr>
<td></td>
<td>GEQ</td>
<td></td>
<td>$S, x, y \rightarrow S, x \geq y$</td>
</tr>
<tr>
<td></td>
<td>EQ</td>
<td></td>
<td>$S, x, y \rightarrow S, x = y$</td>
</tr>
<tr>
<td></td>
<td>NEQ</td>
<td></td>
<td>$S, x, y \rightarrow S, x \neq y$</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>NOT</td>
<td></td>
<td>$S, x \rightarrow S, \sim x$</td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td></td>
<td>$S, x, y \rightarrow S, x \wedge y$</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td></td>
<td>$S, x, y \rightarrow S, x \vee y$</td>
</tr>
<tr>
<td><strong>Bitwise</strong></td>
<td>SHL</td>
<td></td>
<td>$S, x, y \rightarrow S, x \text{ shifted left by } y \text{ bits}$</td>
</tr>
<tr>
<td></td>
<td>SHR</td>
<td></td>
<td>$S, x, y \rightarrow S, x \text{ shifted right by } y \text{ bits}$</td>
</tr>
<tr>
<td></td>
<td>BAND</td>
<td></td>
<td>$S, x, y \rightarrow S, x &amp; y$</td>
</tr>
<tr>
<td></td>
<td>BOR</td>
<td></td>
<td>$S, x, y \rightarrow S, x</td>
</tr>
<tr>
<td></td>
<td>BXOR</td>
<td></td>
<td>$S, x, y \rightarrow S, x \oplus y$</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>RJRT</td>
<td>dist</td>
<td>$S \rightarrow S; PC+ = \text{dist if } x$</td>
</tr>
<tr>
<td></td>
<td>RJF</td>
<td>dist</td>
<td>$S \rightarrow S; PC+ = \text{dist if } \sim x$</td>
</tr>
<tr>
<td></td>
<td>JUMP</td>
<td>addr</td>
<td>$S \rightarrow S; PC = \text{addr}$</td>
</tr>
<tr>
<td><strong>Message operations</strong></td>
<td>SPLICE</td>
<td></td>
<td>readies an array for being packed in a message</td>
</tr>
<tr>
<td></td>
<td>SPLICE2</td>
<td></td>
<td>readsies (a section of) an array for being packed in a message</td>
</tr>
<tr>
<td></td>
<td>CREATE_MSG</td>
<td></td>
<td>creates a message</td>
</tr>
<tr>
<td></td>
<td>EXTEND_MSG</td>
<td></td>
<td>extends a message</td>
</tr>
</tbody>
</table>

Table 2.2: A brief description of the OSAS instruction set[3]
Figure 2.4: OSAS opcodes: Encoding

greater than the absolute byte index of the JUMP; and in the case of while loops, \( y \) is strictly less than the absolute byte index of the JUMP.

Figure 2.5 shows a bytecode with RJFs and JUMPs and illustrates their function. The numbers given to the left of the bytecode are absolute byte indexes.

### 2.4.2 OSAS program constructs

OSAS programs are made from a set of constructs. We call these program constructs *blocks*. A block is a set of bytecode instructions grouped together by some common

Figure 2.5: RJFs and JUMPs
characteristics. An OSAS program is made up of four types of blocks: basic blocks, conditional blocks, while blocks, and if-then blocks. We give informal definitions of the blocks below. In Section 2.4.3, we present algorithms to identify and delineate them.

### 2.4.2.1 Basic blocks

A basic block is a sequence of bytecode instructions which has only one entry point, which is at the start, and only one exit point, which is at the end (See Figure 2.6). That is, there are no branches from or into a basic block except possibly the ones at the start and end. A basic block can contain RJF and JUMP instructions as long as they are at the exit. Basic blocks can occur inside the other three types of blocks or on their own.

### 2.4.2.2 Conditional blocks

A conditional block is made up of an if-then-(elif-then)*-else statement (See Figure 2.6). In a conditional block, program control can branch from some points within the block. These branch points are provided by RJF and JUMP instructions.

### 2.4.2.3 While blocks

A while block is made up of a while loop. A while block has one RJF and one JUMP instructions. The JUMP instruction enables looping. The RJF instruction enables exit from the while loop (See Figure 2.6).

### 2.4.2.4 If-then blocks

The if-then statement does not, by definition, form a conditional block because it doesn’t have an else case. Therefore it is a separate type of block on its own. If-then blocks are characterized by having one RJF, which belongs to neither while blocks nor conditional blocks (See Figure 2.6).
2.4.2.5 Example program

Figure 2.6 shows an example program. In the figure, we show a block mapping between
the source and the corresponding bytecode.

2.4.2.6 OSAS bytecode block composition

Figure 2.7 shows the possible block composition for OSAS bytecode. In the figure, a
\textit{body} is an OSAS bytecode. We see that basic blocks don’t contain any other type of
block inside them. With the remaining 3 types of blocks, each block can contain any
combination of the four types of blocks recursively (nesting).

2.4.3 Block identification and delineation

2.4.3.1 Definitions

\textbf{Block identification} is the process of locating blocks in an OSAS bytecode. This
process is guided by the RJF and JUMP instructions in the bytecode. Sections 2.4.3.4
to 2.4.3.7 present algorithms for identifying the four types of blocks.

\textbf{Block delineation} is to determine the exact start and end byte indices of a block after
it has been identified and to dissect it to its building blocks. Sections 2.4.3.9 to 2.4.3.12
present formulas for delineating the four types of blocks.

2.4.3.2 Notations

The absolute position of byte $x$ in a bytecode is given by $BI(x)$ where $BI$ stands for Byte
Index. For an instruction $RJF$ $x$, $BI(x) = BI(RJF) + 1$. Similarly for an instruction
$JUMP$ $y$, $BI(y) = BI(JUMP) + 1$.

For block identification and delineation purposes, we will use the value of some bytes
in the bytecode. To refer to the value of the byte at byte index $x$ we use the notation
$M(x)$. For an instruction $RJF$ $x$, $M(x) = M(BI(RJF)+1)$. Similarly for an instruction
$JUMP$ $y$, $M(y) = M(BI(JUMP) + 1)$. 
Figure 2.6: Example program containing the four types of blocks
To refer to an RJF at byte index $z$, we write $RJF@z$. Similarly to refer to a JUMP at byte index $w$, we write $JUMP@w$.

We use $[a, b]$ to refer to an interval of bytes from byte index $a$ to $b$ inclusive. For instance, the interval $[M(BI(JUMP) + 1), BI(RJF) + 1]$ refers to the bytes from byte index $M(BI(JUMP) + 1)$ to $BI(RJF) + 1$.

### 2.4.3.3 Order for block identification

The order for block identification in a given bytecode is: while and conditional blocks (in any order) followed by if-then blocks and finally basic blocks. We reason out why we have to follow this order in Section 2.4.3.6.

### 2.4.3.4 An algorithm for identifying while blocks in bytecode

\(^2\)Given a bytecode and its absolute byte indices, any pair $(RJF_i, JUMP_j)$ which fulfills the following three characteristics constitutes a while block.

1. $BI(RJF_i) < BI(JUMP_j)$

\(^2\)The presentation of the algorithms is not in the form of pseudocode but rather informal.
2. \( BI(RJF_i) > M(BI(JUMP_j) + 1) \)

3. \( BI(RJF_i) + M(BI(RJF_i) + 1) = BI(JUMP_j) \)

The algorithm runs in linear time with respect to the number of RJF and JUMP pairs. Figure 2.8 illustrates the above three characteristics.

**Figure 2.8**: A pictorial depiction of the algorithm for identifying while blocks (Section 2.4.3.4)

**Example**

From the source and bytecode given in Figure 2.9, we visually identify that the pair \((RJF@5, JUMP@11)\) constitutes a while block.
We now apply the algorithm to the bytecode given in Figure 2.9. In Figure 2.10, all the three characteristics given by the algorithm are shown to be true for the pair \((RJF@5, JUMP@11)\). Therefore, this pair constitutes a while block.

2.4.3.5 An algorithm for identifying conditional blocks in bytecode

Given a bytecode and its absolute byte indexes, we follow the following four steps to identify the conditional blocks. The Second step helps reduce the running time by

\[3\] The presentation of the algorithms is not in the form of pseudocode but rather informally.
grouping JUMPs with the same \( M(BI(JUMP)+1) \) together. This is because JUMPs with different \( M(BI(JUMP)+1) \) values can’t form a conditional block.

**Step 1:** Develop a table which has \#JUMPs rows and two columns. The first column holds the byte indexes of the JUMPs; and the second column holds the \( M(BI(JUMP)+1) \) values for the corresponding JUMPs. Develop a similar table for the RJFs.

**Step 2:** From the table of Step 1, we group the JUMPs with the same \( M(BI(JUMP)+1) \). The maximum \#groups is \#JUMPs.

**Step 3:** For each group from Step 2, we develop sequences as:

\[
S_{i,i} = \#JUMPs \text{ in the group} = (RJF_1, JUMP_1, RJF_2, JUMP_2, \ldots, RJF_i, JUMP_i) \quad (2.1)
\]

For each group, the number of sequences is:

\[
\#S_{i,i} = \#JUMPs \text{ in the group} = (\#RJFs) \times (\#JUMPs) \times (\#RJFs - 1) \\
\times (\#JUMPs - 1) \\
\times \ldots \times (\#RJFs - i + 1) \times (1) \quad (2.2)
\]

**Step 4:** A sequence from Step 4 constitutes a conditional block if and only if the following 5 conditions are fulfilled\(^4\). We later refer to these conditions as Condition 1, Condition 2, \ldots, Condition 5. For all the conditions, we use the index \( i \) from Step 3.

1. For all \( j \leq i, BI(RJF_j) < BI(JUMP_j) \)
2. For all \( k < i, BI(JUMP_k) < BI(RJF_{k+1}) \)
3. For all \( p \leq i, BI(RJF_p) + M(BI(RJF_p) + 1) = BI(JUMP_p) \)
4. For all \( q < i, M(BI(JUMP_q) + 1) > BI(JUMP_i) \)
5. for all \( w \leq i, BI(JUMP_w) < M(BI(JUMP_i) + 1) \)

The first and second conditions reflect on the relative order of RJFs and JUMPs in terms of their absolute byte indexes. The third condition makes sure that an RJF points to

\(^4\)The conditions can be applied in any order; and some orders may be more efficient than others.
its succeeding JUMP; and the fourth condition dictates that all JUMPs should jump beyond the byte index of the last JUMP in the sequence. The fifth condition prohibit looping.

Step 3, where we form the sequences, has exponential running time. Therefore the algorithm for identifying conditional blocks has exponential running time. We observed from the ET analysis tool (Section 3.6) that for any practical size of an OSAS application, the tool runs reasonably fast (Also See Section 2.4.3.8).

**Example**

From the source and bytecode given in Figure 2.11, we visually identify that there are 2 conditional blocks.

We now apply the algorithm to the bytecode given in Figure 2.11.

**Step 1**: See Tables 2.3 and 2.4.

**Step 2**: We have three groups:

- Group 1: JUMP@20
- Group 2: JUMP@22, JUMP@38
- Group 3: JUMP@51, JUMP@62
Figure 2.11: Identifying two conditional blocks visually
Table 2.5: Group 1 sequences

<table>
<thead>
<tr>
<th>S/N</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(RJF@7, JUMP@20)</td>
</tr>
<tr>
<td>2</td>
<td>(RJF@14, JUMP@20)</td>
</tr>
<tr>
<td>3</td>
<td>(RJF@29, JUMP@20)</td>
</tr>
<tr>
<td>4</td>
<td>(RJF@34, JUMP@20)</td>
</tr>
<tr>
<td>5</td>
<td>(RJF@47, JUMP@20)</td>
</tr>
<tr>
<td>6</td>
<td>(RJF@58, JUMP@20)</td>
</tr>
</tbody>
</table>

Table 2.6: A partial view of Group 2 sequences

<table>
<thead>
<tr>
<th>S/N</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(RJF@7, JUMP@22, RJF@14, JUMP@38)</td>
</tr>
<tr>
<td>2</td>
<td>(RJF@14, JUMP@22, RJF@7, JUMP@38)</td>
</tr>
<tr>
<td>3</td>
<td>(RJF@29, JUMP@22, RJF@7, JUMP@38)</td>
</tr>
<tr>
<td>4</td>
<td>(RJF@34, JUMP@22, RJF@7, JUMP@38)</td>
</tr>
<tr>
<td>5</td>
<td>(RJF@47, JUMP@22, RJF@7, JUMP@38)</td>
</tr>
<tr>
<td>6</td>
<td>(RJF@58, JUMP@22, RJF@7, JUMP@38)</td>
</tr>
</tbody>
</table>

Group 1: \{\text{JUMP}@20\}

\textbf{Step 3}: \#JUMPs of Group 1 is 1; and \#RJFs = 6. Therefore we have \#RJFs \times 1 = 6 \times 1 = 6 possible \(s_1\) sequences, given by Table 2.5.

\textbf{Step 4}: After applying Condition 1, only two sequences remain: (RJF@7, JUMP@20), (RJF@14, JUMP@20). Condition 2 is not applicable to this group. After applying Condition 5, only one sequence remains: (RJF@7, JUMP@20). After applying Condition 3, nothing remains. Therefore none of the sequences in Group 1 constitutes a conditional block. This can be confirmed from Figure 2.11.

Group 2: \{\text{JUMP}@22, \text{JUMP}@38\}

\textbf{Step 3}: \#JUMPs for Group 2 = 2. Therefore we have: \(#RJFs \times 2 \times (\#RJFs - 1) \times 1 = 6 \times 2 \times 5 \times 1 = 60\) possible \(s_2\) sequences. For space reasons we only show a partial view of the 60 sequences (Table 2.6).

\textbf{Step 4}: After applying just Conditions 1, 2, and 5, we remain with only 3 sequences:
(RJF@7,JUMP@22,RJF@29,JUMP@38)

(RJF@7,JUMP@22,RJF@34,JUMP@38)

(RJF@14,JUMP@22,RJF@34,JUMP@38)

After applying Conditions 3 and 4, only one sequence remains:
(RJF@7,JUMP@22,RJF@29,JUMP@38).

Since (RJF@7,JUMP@22,RJF@29,JUMP@38) satisfied all five conditions, it constitutes a conditional block. We can confirm from Figure 2.11 that this sequence constitutes a conditional block.

Group 3: \{JUMP@51,JUMP@62\}

**Step 3:** \#JUMPs for Group 3 = 2. Therefore we have: 

\[ (#RJFs \times 2 \times (#RJFs - 1) \times 1 = 6 \times 2 \times 5 \times 1 = 60) \]

possible \(s_2\) sequences. They can be constructed in a similar manner to those shown in Table 2.6.

**Step 4:** After applying Conditions 1, 2, and 5, only five sequences remain:

1. (RJF@7,JUMP@51,RJF@58,JUMP@62)
2. (RJF@14,JUMP@51,RJF@58,JUMP@62)
3. (RJF@29,JUMP@51,RJF@58,JUMP@62)
4. (RJF@34,JUMP@51,RJF@58,JUMP@62)
5. (RJF@47,JUMP@51,RJF@58,JUMP@62)

After applying Conditions 3 and 4, only one sequence remains:
(RJF@47,JUMP@51,RJF@58,JUMP@62). Figure 2.12 illustrates the test for Condition 3 on the five sequences.

Since (RJF@47,JUMP@51,RJF@58,JUMP@62) satisfied all five conditions, it constitutes a conditional block. This can be confirmed from Figure 2.11.
Figure 2.12: Test for condition 3 on the five sequences
2.4.3.6 An algorithm for identifying if-then blocks in bytecode

In Section 2.4.3.3, “Order for block identification”, it was stated that the order for block identification in a given bytecode is: while and conditional blocks (in any order) followed by if-then blocks and finally basic blocks. The reason for this order is that identifying if-then blocks depends on which RJFs belong to while and conditional blocks. After identifying all while and conditional blocks in bytecode, the remaining RJFs belong to if-then blocks. An if-then block contains a single RJF. Note that there are no JUMPs left after all while and conditional blocks have been identified.

Therefore, the algorithm for identifying an if-then blocks is: After identifying all while and conditional blocks in a bytecode, the remaining RJFs constitute if-then blocks. The algorithm runs in linear time with respect to the number of RJFs.

2.4.3.7 An algorithm for identifying basic blocks in bytecode

After identifying all while, conditional, and if-then blocks, the remaining parts of the bytecode are basic blocks. The algorithm runs in linear time with respect to the size of the bytecode.

2.4.3.8 A note on the efficiency of the algorithms

There could be many other ways of identifying the while, conditional, if-then, and normal blocks in a bytecode. We chose the most ‘apparent’ ones. We refer the complete algorithm-space exploration to find the most efficient algorithms to a further work.

2.4.3.9 Delineating while blocks

A while block has a test and a body; and the test executes once more than the body. The test and body of a while loop identified by the pair \((RJF_i, JUMP_j)\) are delineated as: (Also See Figure 2.13)

While block test delineation: while block test=\([M(BI(JUMP_j) + 1), BI(RJF_i) + 1]\)

While block body delineation: while block body=\([BI(RJF_i) + 2, BI(JUMP_j) + 1]\)
Chapter 2. *Measurement-based Execution Time (ET) Model*

The Worst-case Execution Time (WCET) and Best-case Execution Time (BCET) of a while block are given in Figure 2.13. For BCET, the while loop has zero iterations; and therefore the BCET only contains a single execution of the test.

![Figure 2.13: Delineating while blocks](image)

**2.4.3.10 Delineating conditional blocks**

A conditional block is a composition of tests and bodies. Figure 2.14 shows the general structure of an if-then-(elif-then)*-else construct. Note from the figure that the number of bodies is one larger than the number of tests. This is due to the else case. Figure 2.14 also shows the possible control flows in an if-then-(elif-then)*-else construct.

Taking a conditional block identified (See Section 2.4.3.5) with the sequence $(RJF_1, JUMP_1, RJF_2, JUMP_2, \ldots, RJF_n, JUMP_n)$, its bodies and tests are delineated

39

Figure 2.14: If-then(elif-then)*-else constructs: General structure

as:

\[
body_i = \begin{cases} 
[BI(RJF_i) + 2, BI(JUMP_i) + 1], & \text{for } 1 \leq i \leq n \\
[BI(JUMP_{i-1}) + 2, M(BI(JUMP_{i-1}) + 1) - 1], & \text{for } i = n + 1
\end{cases}
\tag{2.3}
\]

\[
test_i = [BI(JUMP_i) + 2, BI(RJF_{i+1}) + 1], \quad 1 \leq i \leq n - 1
\tag{2.4}
\]

The sequence of RJFs and JUMPs is in a conditional block (See Section 2.4.3.5) is: \( RJF_1 \rightarrow JUMP_1 \rightarrow RJF_2 \rightarrow JUMP_2 \rightarrow \ldots \rightarrow RJF_n \rightarrow JUMP_n \). The sequence of bodies and tests in a conditional block is: \( body_1 \rightarrow test_1 \rightarrow body_2 \rightarrow test_2 \rightarrow \ldots \rightarrow body_{n-1} \rightarrow test_{n-1} \rightarrow body_n \rightarrow body_{n+1} \). Note that a conditional block, as we define it here, starts with a body although an if-then-(elif-then)*-else construct starts with a test. There are two reasons for leaving out the first test and letting it be considered as a basic block. The first reason is that the first test is executed always (See Figure 2.14); and so we can leave it for it to be considered as a basic block. The second reason is that it is difficult to delineate the first test; and this is because there is no RJF or JUMP which is part of the conditional block but comes before the first test. For these reasons we leave the first test of a conditional statement for it to be considered as a basic block. The decision to leave out the first test for it to be considered as a basic block is made only for analytical convenience and does not alter the ET value of a bytecode. We get
the same ET value whether we consider the first test as part of the conditional block or we leave it out for it to be considered as a basic block.

The WCET of a conditional block is the maximum of the ET of all sequences of tests (including empty) followed by a body. We define case to be a sequence of tests followed by a body; and we have \( n + 1 \) cases for a conditional block with \( n + 1 \) bodies and \( n - 1 \) tests. The BCET of a conditional block is the minimum of the ET of all cases.

Figure 2.15, using an example program, illustrations the conditional block delineation process. Figure 2.16 takes the result of Figure 2.15 and shows how to construct the cases and calculate the WCET and BCET of a conditional block.

### 2.4.3.11 Delineating if-then blocks

An if-then block has a test and a body (See Figure 2.17). We leave out the test of an if-then statement from being considered as part of the if-then block for the same reason we leave out the first test of a conditional statement. We leave out the test for it to be considered as a basic block. Therefore we only delineate the body. Note again that the decision to leave out the test for it to be considered as a basic block is made only for analytical convenience and does not alter the ET value of a bytecode. We get the same ET value whether we consider the test as part of the if-then block or we leave it out for it to be considered as a basic block.

An if-then block corresponding to \( RJF_i \) has its body delineated as:

\[
\text{body} = [BI(RJF_i) + 2, BI(RJF_i) + M(BI(RJF_i) + 1) + 1]
\]

Figure 2.17 shows, through an example, how to delineate an if-then block and determine its WCET and BECT.

The BCET of an if-then block occurs when its test evaluates to false (that is, its body doesn’t execute); and since, by definition, an if-then block is made up of only its body, the BCET of an if-then block is zero.
2.4.3.12 Delineating basic blocks

After all while, conditional, and if-then blocks of a given bytecode are delineated, what is left constitutes basic blocks.

2.4.3.13 A note on the tests of while, conditional, and if-then blocks

While, conditional, and if-then blocks contain tests. All these tests are basic blocks. That is, there are no while, conditional, or if-then blocks inside tests.

Figure 2.16: Delineating conditional blocks: Determining WCET and BCET

\[ WCET = \max \left( \text{ET(body}_1\text{)}, \ \text{ET(test}_1\text{ + body}_2\text{)}, \ \text{ET(test}_1\text{ + test}_2\text{ + body}_3\text{)}, \ \text{ET(test}_1\text{ + test}_2\text{ + body}_4\text{)} \right) \]

\[ BCET = \min \left( \text{ET(body}_1\text{)}, \ \text{ET(test}_1\text{ + body}_2\text{)}, \ \text{ET(test}_1\text{ + test}_2\text{ + body}_3\text{)}, \ \text{ET(test}_1\text{ + test}_2\text{ + body}_4\text{)} \right) \]
2.4.3.14 Summary: Block identification and delineation

Figure 2.18 summarizes the block identification and delineation process. In the figure, it is not necessary that the tasks be done in that particular order; but the block identification order given in Section 2.4.3.3 has to be observed. The block identification and delineation process is iterated (due to nesting) until there are no bodies left, at which point the WCET and BCET of the original bytecode are determined. Recall that a body is an OSAS bytecode, and it is some combination of blocks. The block identification and delineation process stops when all bodies have been broken down to basic blocks.

2.4.4 Determining the ET of an OSAS program

The ET of an OSAS bytecode is made up of ET of its blocks and ET due to interpreter overheads. We have four types of overheads: $iT_i$Overhead, $eOverhead$, $eTeOverhead$, $eTeOverhead$. 

---

**Figure 2.17:** Delineating if-then blocks: Example
and \( f_{Overhead} \). \( iT_{Overhead} \) is the interpreter’s instruction to instruction overhead. \( e_{Overhead} \) is the interpreter’s EG-startup and finalization overhead. \( eTe_{Overhead} \) is the interpreter’s EG-to-EG overhead. \( f_{Overhead} \) is the interpreter’s function-call overhead. The values of these overheads is given in Section 2.6.2.3.

The ET of the execution units is (Also see Figure 2.19):

\[
ET_{function} = ET_{blocks} + f_{Overhead}
\]
\[
ET_{EG} = ET_{blocks} + e_{Overhead}
\]
\[
ET_{EG\ chain} = ET_{EGs} + (\#EGs - 1) \times eTe_{Overhead}
\]
\[
ET_{handler} = ET_{blocks}
\]
\[
ET_{CBA} = ET_{blocks}
\] (2.5)

\(iT\text{Overhead}\) is considered within \(ET_{\text{blocks}}\). \(ET_{\text{block}}\) is the sum of the ETs of the each instruction in the block.

See Figure 2.27 for a time line showing the different types of overheads.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure219.png}
\caption{Breakdown of ET of execution units}
\end{figure}

2.4.4.1 Determining the WCET and BCET of a bytecode: Example

In this section, we show the process of determining the WCET and BCET of a given bytecode through an example. Figure 2.20 shows a source and its bytecode; and Figures 2.21 and 2.22 show the WCET and BCET determination process, respectively. The WCET has \#iterations of while loops as parameter; and the user has to supply these values (Also see Section 4.2.3).

2.5 Platform: TelosB running MANTIS Operating System

We used TelosB nodes running the MANTIS Operating System for the measurements. TelosB nodes\[40\] use the MSP430 MCU.

2.5.1 MSP430

MSP430\[41\] is a micro-controller unit (MCU) from Texas Instruments. It contains a 16-bit RISC architecture CPU, which runs at 3.7 MHz in the version of TelosB we used\[40\]. MPS430 is mostly used with low power and low cost embedded applications.
Figure 2.20: Determining the ET of a bytecode: source and bytecode

**Figure 2.21:** Determining the ET of a bytecode: WCET

The diagram shows the process of determining the execution time (ET) of bytecode instructions. The steps include identifying blocks, determining the execution time of each block, and calculating the total ET using the WCET formula:

\[ \text{WCET} = \text{ET}(BB1) + \text{ET}(BB2) \times \#\text{trns} + \text{ET}(BB3) \times \#\text{trns} + \text{ET}(BB4) + \text{max}(\text{ET}(BB5), \text{ET}(BB6+BB7), \text{ET}(BB6+BB8)) + \text{ET}(BB8) + \text{ET}(BB10) + \text{ET}(BB11) + \text{ET}(BB12) \]

Figure 2.22: Determining the ET of a bytecode: BCET
2.5.2 Mantis Operating System

MANTIS (See Figure 2.23) is a multi-threaded cross-platform embedded operating system for wireless sensor networks [42, 43]. The threads are scheduled using a round-robin scheduling algorithm. MANTIS is a preemptive system. That is, there are more than one thread active at a time; and these threads preempt each other according to their priorities.

![MANTIS Operating System Diagram](image)

Figure 2.23: The MANTIS Operating System

2.5.3 The OSAS thread

The tasks within the OSAS thread are executed by cooperative multitasking. The tasks are subscription activations and received messages (actions). (See Figure 2.24).

The subscriptions within the OSAS thread are ordered by their next activation time. The OSAS runtime has two queues, one for subscriptions (sorted by the next activation time) and one for incoming messages (first-in-first-out). When the periodic timer fires, an OSAS clock counter is increased and a counting semaphore is used to activate the OSAS thread. When a message arrives, it is put into the message queue and the same counting semaphore is used. After the OSAS thread is activated, it will first process all the subscriptions that have to be executed at the current time, which returns immediately when nothing needs to be done. Then, it will process one incoming message from the queue. After that, it will wait for the semaphore again. Since it is a counting semaphore, all messages will be processed as fast as possible, while being interleaved...
with the subscriptions where required. In case no counting semaphores are available, messages would be processed during the next 10ms\(^5\) interval.

### 2.5.4 Timer interrupt

MANTIS is driven by a 1KHz clock. The execution time of the timer interrupt routine is 36\(\mu\)s, at 3.7 MHz. That means 3.6% of the CPU time is spent handling the timer interrupt. This is a constant overhead.

\(^5\)The unit of time in OSAS is 10ms.
2.6 Measurements

2.6.1 Measurement setup

To measure the execution time of each bytecode instruction, the interpreter was instrumented with LED toggles (see Figure 2.25). The LED toggles mainly mark the beginning and end of execution of BC instructions. Then an oscilloscope (MS07102T) is used to measure the ET of each bytecode instruction. From these measurements a lookup data is developed which is used to determine the ET of basic blocks.

![Figure 2.25: Measurement setup](image)

2.6.1.1 Interpreter instrumentation

The interpreter is instrumented such that LEDs are toggled to mark the start and end of execution of BC instructions. Two LEDs are used: MAIN LED and AUXILIARY LED. The section of the interpreter with the instrumentation is shown in Listing 2.1 (showing only the relevant parts).

```c
// Assume the bits of the BC instruction are numbered 7-0
while() {
    MAIN LED ON
    switch(766) {
        case ---:
            switch(43) {
                case PUSHL, PUSHG:
                case STORE:
            }
        case PUSHC:
```
case EXTEND:
    other ...
MAIN and AUXILIARY LEDs ON

case --: switch (5-0) // special instructions like FLOOD, ADD, SUB, MUL, MOD, DIV
{
    case FLOOD:
    case ADD:
    case SUB:
    ...
}
All LEDs OFF

Listing 2.1: Interpreter LED instrumentation

From Listing 2.1, we observe that every executed BC instruction is bounded by the ON and OFF of the MAIN LED. For some bytecode instructions, it is convenient to use a separate LED to bound a certain section of their execution time; and that is what the AUXILIARY LED is used for. Figure 2.26 demonstrates the measurement procedure through an example application.

2.6.2 The lookup data

We now present the execution time lookup data for the OSAS bytecode instructions. It was built using TelosB nodes with MSP430 MCU running at 3.7 MHz.

We will present the lookup data in three sections:
1. A table of the execution time of the 44 bytecode instructions

2. A child table for CALL[id]

3. A table of execution time of administrative tasks

2.6.2.1 The execution time of the 44 Bytecode instructions

See Table 2.7.
<table>
<thead>
<tr>
<th>S/N</th>
<th>BC instruction</th>
<th>Execution time in µs; precision=1µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PUSHV</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>PUSHL</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>PUSHA</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>PUSHG</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>STOREV</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>STOREL</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>STOREA</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>STOREG</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>VERB</td>
<td>91</td>
</tr>
<tr>
<td>10</td>
<td>VERB16</td>
<td>43</td>
</tr>
<tr>
<td>11</td>
<td>PUSHC</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>EXTEND</td>
<td>26</td>
</tr>
<tr>
<td>13</td>
<td>CALL</td>
<td>See Table 2.8</td>
</tr>
<tr>
<td>14</td>
<td>POP</td>
<td>33</td>
</tr>
<tr>
<td>15</td>
<td>PUSHIS</td>
<td>41</td>
</tr>
<tr>
<td>16</td>
<td>STORES</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>NOT</td>
<td>35</td>
</tr>
<tr>
<td>18</td>
<td>AND</td>
<td>40</td>
</tr>
<tr>
<td>19</td>
<td>OR</td>
<td>39</td>
</tr>
<tr>
<td>20</td>
<td>LESS</td>
<td>39</td>
</tr>
<tr>
<td>21</td>
<td>MORE</td>
<td>39</td>
</tr>
<tr>
<td>22</td>
<td>LEQ</td>
<td>39</td>
</tr>
<tr>
<td>23</td>
<td>GEQ</td>
<td>39</td>
</tr>
<tr>
<td>24</td>
<td>EQ</td>
<td>39</td>
</tr>
<tr>
<td>25</td>
<td>NEQ</td>
<td>39</td>
</tr>
<tr>
<td>26</td>
<td>ADD</td>
<td>36</td>
</tr>
<tr>
<td>27</td>
<td>SUB</td>
<td>36</td>
</tr>
<tr>
<td>28</td>
<td>MUL</td>
<td>41</td>
</tr>
<tr>
<td>29</td>
<td>DIV</td>
<td>94</td>
</tr>
<tr>
<td>30</td>
<td>MOD</td>
<td>94</td>
</tr>
<tr>
<td>31</td>
<td>SHL</td>
<td>56</td>
</tr>
<tr>
<td>32</td>
<td>SHR</td>
<td>56</td>
</tr>
<tr>
<td>33</td>
<td>BAND</td>
<td>36</td>
</tr>
<tr>
<td>34</td>
<td>BOR</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>BXOR</td>
<td>36</td>
</tr>
<tr>
<td>36</td>
<td>NTFY</td>
<td>4491 (worst-case)</td>
</tr>
<tr>
<td>37</td>
<td>FLOOD</td>
<td>5252 (worst-case)</td>
</tr>
<tr>
<td>38</td>
<td>CREATE_MSG</td>
<td>560</td>
</tr>
<tr>
<td>39</td>
<td>EXTEND_MSG</td>
<td>86</td>
</tr>
<tr>
<td>40</td>
<td>RJT</td>
<td>29</td>
</tr>
<tr>
<td>41</td>
<td>RJF</td>
<td>29</td>
</tr>
<tr>
<td>42</td>
<td>JUMP</td>
<td>29</td>
</tr>
<tr>
<td>43</td>
<td>SPLICE</td>
<td>438</td>
</tr>
<tr>
<td>44</td>
<td>SPLICE2</td>
<td>430</td>
</tr>
</tbody>
</table>

Table 2.7: The execution time of the 44 Bytecode instructions
2.6.2.2 A child table for CALL[id]

Table 2.8 presents the ET for the CALL[id] BC instructions, which correspond to system functions. System functions, unlike user-defined functions, are available together with the firmware. Note from the table that system functions Median, Avg, AvgSqr, and Sqrt have their best-case execution times reported. This is because all these four system functions take parameters which cannot be upper bounded. We attempted to parameterize their execution time, but we didn’t recognize any pattern.
<table>
<thead>
<tr>
<th>Administrative task</th>
<th>Execution time in μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer interrupt</td>
<td>36</td>
</tr>
<tr>
<td>Instruction to instruction transition</td>
<td>13</td>
</tr>
<tr>
<td>LED toggling overhead</td>
<td>2</td>
</tr>
<tr>
<td>EG start overhead</td>
<td>19</td>
</tr>
<tr>
<td>EG finalization overhead</td>
<td>12</td>
</tr>
<tr>
<td>User defined function start up overhead</td>
<td>157</td>
</tr>
<tr>
<td>User defined function finish up overhead</td>
<td>97</td>
</tr>
<tr>
<td>Temp sensor access</td>
<td>260000</td>
</tr>
<tr>
<td>Acceleration X sensor access</td>
<td>80000</td>
</tr>
<tr>
<td>Acceleration Y sensor access (dummy)</td>
<td>0</td>
</tr>
<tr>
<td>Acceleration Z sensor access (dummy)</td>
<td>0</td>
</tr>
<tr>
<td>EG to EG overhead</td>
<td>83</td>
</tr>
<tr>
<td>Message reception</td>
<td>674</td>
</tr>
<tr>
<td>Processing of CBA upon message reception</td>
<td>253</td>
</tr>
<tr>
<td>Message processing (end) to message retransmit (start)</td>
<td>777</td>
</tr>
<tr>
<td>Message retransmit</td>
<td>234</td>
</tr>
<tr>
<td>Message latency (1 hop)</td>
<td>58000</td>
</tr>
</tbody>
</table>

Table 2.9: A table of execution time of administrative tasks; precision=1μs

2.6.2.3 A table of execution time of administrative tasks

See Table 2.9.
From Table 2.9, we determine the overheads we listed in Section 2.4.4. Figure 2.27 shows a time line for the overheads.

\[
\text{fOverhead} = \text{start overhead} + \text{finalization overhead} = 157 + 97 = 254 \text{ microseconds}
\]

\[
\text{eOverhead} = \text{start overhead} + \text{finalization overhead} = 19 + 12 = 31 \text{ microseconds}
\]

\[
\text{eTeOverhead} = 83 \text{ microseconds}
\]

\[
\text{iTiOverhead} = 13 \text{ microseconds}, \text{ and it is considered within the ET of an instruction.}
\]

For \( n \) instructions of a basic block, there are \( n - 1 \) iTiOverheads.

Figure 2.27: Time line of ET of execution units
2.6.2.4 LED toggling overhead

From Table 2.9, we see that the LED toggling overhead is 2 microseconds, which is at most one-sixth the ET of any bytecode instruction. This is because it is done in the interpreter itself rather than in an OSAS application. Therefore the LED toggling overhead is ignored.

2.7 ET analysis tool

A ET analysis tool was developed out of the ET model presented in this chapter. The tool is written in C++, and it has above five thousand (5000) lines-of-code. The tool takes a wbc file as an input and it gives the ET breakdown, WCET, and BCET values. The tool has two main parts (Figure 2.28). Process-wbc-File breaks up the wbc file to execution units and prepares the bytecode in a certain format suitable for the second part. The second part, Analyze-Bytecode, reads the execution units one by one and produces their ETs. We demonstrate the use of the tool though an example.

![Figure 2.28: ET model tool](ET_model_tool)

2.7.1 Using the ET analysis tool: Example

Figure 2.29 presents an application, its wbc (after it is processed by Process-wbc-File) and the final output of the tool (the ET of each of the execution units).

From Figure 2.29, it can be seen that EG $e1$ has WCET of $471 \times 10 + 1651 = 6361$ microseconds, filling in 10 for $\#itrns1$. The BCET is 1248 microseconds.

Figure 2.29: ET analysis tool
2.7.2 Requirements of the tool

The tool has some requirements. The first one is that there should be no block nesting. That is, no conditional or if-then blocks inside while blocks, no while or if-then blocks inside conditional blocks, no while or conditional blocks inside if-then blocks. This limitation was introduced for two reasons. First, the event-based nature of OSAS provides looping; and so nesting of while loops inside while loops is not common. By looking through WASP project OSAS application files, we found out that 86% of them have no nesting. Secondly, the tool already has above 5000 lines-of-code; and introducing nesting capabilities will increase its size significantly. Therefore, this task is referred to a further work (See Section 4.2.4). But the tool is implemented in such a way that it is convenient to extend it with the capability of nesting. To help this extension, the Process-wbc-File part of the tool extracts all nesting information and provides them as output files. Note that EGs should have their conditions set to True. This is because EG conditions result to an if-then block where the test is the EG condition and the body is the EG body. That means if the EG condition is an expression other than True, you can’t have while, conditional, and if-then blocks inside the EG body due to requirement that there should no be block nesting. The inaccuracy introduced by making the EG condition as True is insignificant as the EG condition (test) is usually insignificant (see Section 2.8) compared to the EG body. The second requirement is that if there are user-defined functions, the tool should be run two times: first to get the ET of the functions and manually update the lookup table; and second to get the ET of the execution units which called these user-defined functions. Note that you have to add the \( f_{Overhead} \) to the value of ET obtained for a user-defined function. Also note that the use of user-defined functions is discouraged as each user-defined function call spawns an instance of the interpreter. This leads to larger interpreter overheads.

2.8 ET Model evaluation

The model was evaluated through 20 example applications, some of which taken from the WASP project. The result of the evaluation is shown in Figure 3.10. From the figure, we see that WCET is larger than the measured ET and the BCET is smaller than the measured ET. In the case of the BCET and measured ET, it can be the case that the
BCET is larger than the measured ET. This is due to NTFY and FLOOD instructions of which the lookup data only contains their worst-case execution time value. The ET of NTFY and FLOOD depends on the length of the message. Measurements were taken to see if the ET of these instructions can be parameterized with the message length, but no pattern was observed. Another option is to use the best-case execution time values for the BCET calculations, although the tool doesn’t include this feature.

**Figure 2.30: ET model evaluation**
Chapter 3

Memory model

3.1 Introduction

This chapter presents the memory model for OSAS software components. We start with an introduction to linked lists, which are the data structure used by the MANTIS MMU. We then present the memory models\(^1\). Afterwards we present a memory analysis tool and evaluate the models.

For every application to be uploaded, the memory model answers the question: How much memory\(^2\) will an application or its parts use? The answer to the previous question is then combined with the knowledge of the current memory status to answer the question: Does the application fit in the target nodes in terms of memory requirements?

3.2 Linked-lists

A linked list is a data structure which consists of blocks such that each block has a link to the next (previous) block in the sequence. There are two types of linked lists; singly and doubly linked lists. With doubly linked lists, each block is not only linked to the next block but also to the previous one (See Figure 3.1). The MANTIS MMU uses doubly-linked lists and allocations are done through the best-fit allocation algorithm\[^{[44, 45]}\].

---

\(^1\)Depending on which overheads we consider, we have 4 memory models (see Section 3.4)

\(^2\)This memory usage does not include the run-time (operational) memory usages such as stack and communication buffers as these are statically allocated.
With the best-fit allocation algorithm, an allocation is placed in the smallest block of free memory in which it will fit.

**Figure 3.1:** Linked lists

In Figure 3.1, we see blocks and block headers. A block is a contiguous portion of memory. It can be free or used. When a portion of memory is used, we call it allocated memory block. The used and free blocks are organized in separate linked lists. Figure 3.2 shows a hierarchical structure of the MMU; which has a two-layer administration. One layer constitutes the general administration which is for all linked lists. The second layer is block administration. Adjacent free blocks are merged in a process called defragmentation.

**Figure 3.2:** The hierarchical structure of the MMU

The general administration consists of pointers to the head of the linked lists.

The MANTIS MMU block header is of size 6B and contains:
• The Next pointer (2B): It holds the start address of the next block in the linked list.

• The Previous pointer (2B): It holds the start address of the previous block in the linked list.

• The size of the block (2B): The knowledge of block size is important as the MMU needs to know where the block ends.

Next we discuss two issues with memory management: Internal and external fragmentation overheads. We define an allocation unit to be a predefined size of which all allocation are a multiple of. The allocation unit for the MANTIS MMU is 2B.

### 3.2.1 Internal fragmentation overhead

A memory allocation request is not necessarily a multiple of the allocation unit. However, the MMU allocates memory in multiples of the allocation unit. The unused memory introduced by this phenomenon is called internal fragmentation overhead (see Figure 3.3). For each allocation, the amount of internal fragmentation overhead is strictly less that the allocation unit.

### 3.2.2 External fragmentation overhead

We recall that the MANTIS MMU keeps the free blocks in a linked list. We also recall that the MANTIS MMU has a block header of size 6B. When the MANTIS MMU executes an allocation request, it first tries to locate a free block using the best-fit allocation algorithm. If it has found such a free block, it determines, for this free block, the size of the remaining free block when the allocation is done. If the size of the remaining free block is 6B or less, it adds it to the allocation. This is because a free block of size less than 6B cannot be included in the free list as the block header itself is 6B. For a free block size of 6B, it is possible to keep it in the free list but since the block header will use these 6 Bytes, there will be zero memory left in that block. Therefore the MANTIS MMU includes all free blocks of size 6B or less to the current allocation. We call this external fragmentation overhead. For most allocation, there is no or little external fragmentation overhead. This overhead is included to produce
worst-case memory usage models. Figure 3.3 shows what happens when an allocation leads to a free block of size less than or equal to 6B.

![Diagram showing memory allocation and fragmentation](image)

**Figure 3.3:** Internal and External fragmentation overheads: Example
3.3 Memory layout of OSAS

Figure 3.4 shows the OSAS memory layout. Table 3.1 gives the breakdown of the header sizes. This information was retrieved from the MANTIS and OSAS sources.

![OSAS Memory Layout Diagram](image)

**Figure 3.4:** OSAS memory layout
3.4 The memory models

Depending on which of internal and/or external fragmentation we consider, we have four models as shown in Table 3.2. The input to all the models is the wbc file; and the output is the memory analysis results. The static part of the models is the OSAS memory layout given in figure 3.4. The dynamic parts are the internal and external fragmentation overheads.

From the wbc file, every model extracts the following information: Bytecode sizes, state sizes, and number of parameters (See Figure 3.4). Bytecode sizes are for CBAs, functions, EGs, and actions. State sizes correspond to states of services. Number of parameters correspond to subscriptions.

The memory usage of an OSAS program according to Model 1 is simply what is given in the OSAS memory layout we presented in Figure 3.4.

In Model 2, an external fragmentation overhead of 6B is assumed for every allocation.

In Model 3, for a part with memory usage of \( x \) Bytes, we take internal fragmentation into consideration by rounding up \( x \) to the next multiple of 2.

Model 4 combines the considerations in Models 2 and 3.

As Models 2 and 4 take external fragmentation overhead, which happens very infrequently, into consideration for every allocation, they have large over-estimation. Model 3 has the closest estimate of all since it considers internal fragmentation overhead, which happens for every allocation. See Section 3.7, “Memory model evaluation” where we confirm the these claims using the memory analysis tool.

3.5 Measurements

In order to do measurements, we need a base scenario. We have the following files for our base scenario:

- Scenario.wnl
- InstallGuard.wsp
<table>
<thead>
<tr>
<th>Header</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CBA header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>The next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>CBA ID: 2B</td>
</tr>
<tr>
<td></td>
<td>BC length: 2B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>CBA header = 12B</td>
</tr>
<tr>
<td><strong>Service header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>The next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>Service ID: 1B</td>
</tr>
<tr>
<td></td>
<td>Asynchronous calls: 1B</td>
</tr>
<tr>
<td></td>
<td>Item count: 1B</td>
</tr>
<tr>
<td></td>
<td>Finalized?: 1B</td>
</tr>
<tr>
<td></td>
<td>Pointer to EG chain: 2B</td>
</tr>
<tr>
<td></td>
<td>Number of arguments: 2B</td>
</tr>
<tr>
<td></td>
<td>Array offset: 2B</td>
</tr>
<tr>
<td></td>
<td>State size: 2B</td>
</tr>
<tr>
<td></td>
<td>Pointer to State: 2B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>Service header = 22B</td>
</tr>
<tr>
<td><strong>Function header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>The next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>Function ID: 2B</td>
</tr>
<tr>
<td></td>
<td>BC length: 2B</td>
</tr>
<tr>
<td></td>
<td>Pointer to service: 2B</td>
</tr>
<tr>
<td></td>
<td>Number of arguments: 1B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>Function header = 15B</td>
</tr>
<tr>
<td><strong>EG header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>Next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>EG ID: 2B</td>
</tr>
<tr>
<td></td>
<td>BC length: 2B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>EG header = 12B</td>
</tr>
<tr>
<td><strong>Action header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>Next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>Action ID: 2B</td>
</tr>
<tr>
<td></td>
<td>BC length: 2B</td>
</tr>
<tr>
<td></td>
<td>Stat counter: 2B</td>
</tr>
<tr>
<td></td>
<td>Pointer to service: 2B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>Action header = 16B</td>
</tr>
<tr>
<td><strong>Subscription header</strong></td>
<td>MMU header: 6B</td>
</tr>
<tr>
<td></td>
<td>The next pointer: 2B</td>
</tr>
<tr>
<td></td>
<td>Subscription ID: 2B</td>
</tr>
<tr>
<td></td>
<td>Quality of service: 1B</td>
</tr>
<tr>
<td></td>
<td>Deadline: 1B</td>
</tr>
<tr>
<td></td>
<td>Handler: 1B</td>
</tr>
<tr>
<td></td>
<td>Minimum period: 1B</td>
</tr>
<tr>
<td></td>
<td>Maximum period: 1B</td>
</tr>
<tr>
<td></td>
<td>Parameter count: 1B</td>
</tr>
<tr>
<td></td>
<td>Period directly accessible from bytecode: 2B</td>
</tr>
<tr>
<td></td>
<td>Subscriber’s ID directly accessible from bytecode: 2B</td>
</tr>
<tr>
<td></td>
<td>Time for next check: 2B</td>
</tr>
<tr>
<td></td>
<td>Pointer to service: 2B</td>
</tr>
<tr>
<td></td>
<td>===============</td>
</tr>
<tr>
<td></td>
<td>Subscription header = 24B</td>
</tr>
</tbody>
</table>

Table 3.1: OSAS header sizes
Chapter 3. Memory model

<table>
<thead>
<tr>
<th>Models</th>
<th>Internal fragmentation</th>
<th>External fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Model 2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Model 3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Model 4</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.2: The memory models

- ReportFreeMemory.wsp
- Scenario.wsp

The scenario.wnl file holds the network layout. The InstallGuard application will enable us to see which services and subscriptions are installed on nodes. The ReportFreeMemory application reports back the memory map. We use the memory map information to compute the actual memory usage and in turn use the actual memory usage to validate the output of the models.

We now provide the files (Listings 3.1 to 3.4).

---

2 14 LoaderNode1 LoaderNode2
2 15 CommonNodes Gateway

Listing 3.1: Scenario.wnl

```plaintext
service InstallGuard($Handler)
  on event e1 when True do
    SendToSubscribers($Handler, NodeID(), 64, ListStatus(254));
    SendToSubscribers($Handler, NodeID(), 128, ListStatus(253))
```

subscription InstallGuardSubscription
to InstallGuard($Handler=MonitorNode)
with (period=5s, deadline=1m, send="Normal", exec="Normal")

Listing 3.2: InstallGuard.wsp
Chapter 3. Memory model

service ReportFreeMemory($Handler)
    on event e1 when True do
        SendToSubscribers($Handler, NodeID(), ListStatus(250))

subscription ReportFreeMemorySubscription
    to ReportFreeMemory($Handler=print)
    with (period=5s, deadline=1m, send="Normal", exec="Normal")
.

Listing 3.3: ReportFreeMemory.wsp

import InstallGuard
import ReportFreeMemory

for [Network|*|NodeID()=48]
    install InstallGuard
    install ReportFreeMemory

for [Network|*|NodeType()="LoaderNode"]
    install InstallGuardSubscription on [Network|*|NodeID()=48]

for [Network|*|NodeType()="Gateway"]
    install ReportFreeMemorySubscription on [Network|*|NodeID()=48]
.

Listing 3.4: Scenario.wsp

3.5.1 Measurement setup

This measurement was done with BSN Development Kit v3 nodes running MANTIS.
The layout for the scenario is shown in Figure 3.5.

Figure 3.5: The layout of the scenario

Figure 3.6 shows the Loader and Simulator before the base scenario is installed.
Figure 3.6: The Loader and Simulator before the base scenario is installed

Figure 3.7 shows the Loader and Simulator when the base scenario is running.

From Figure 3.7 we see that the base memory map is (4514, 4564, 2). The first value is the size of the largest free block. The middle value is the size of free memory; and the last value is the number of free blocks (See Figure 3.8).

Until now we were establishing a base scenario to enable us to monitor the node (Node 48). Now we install Service1 (Listing 3.5).

```plaintext
service Service1($Handler, $eventID)
on event e1 when $eventID==1 do
    SendToSubscribers($Handler, NodeID(), Temp(), AccelX(), AccelY(), AccelZ())
subscription Service1Subscription
to Service1($Handler=print, $eventID=1)
with (period=5s, deadline=1m, send="Normal", exec="Normal")
```

Figure 3.7: The Loader and Simulator when base scenario is running

Figure 3.8: Memory map after the base scenario is installed
Chapter 3. Memory model

Listing 3.5: Service1.wsp

We modify Scenario.wsp so that Service1 and Service1Subscription are installed on Node 48 (Listing 3.6).

```plaintext
import InstallGuard
import ReportFreeMemory
import Service1

for [Network|*|NodeID()=48]
    install InstallGuard
    install ReportFreeMemory
    install Service1

for [Network|*|NodeType()="LoaderNode"]
    install InstallGuardSubscription on [Network|*|NodeID()=48]

for [Network|*|NodeType()="Gateway"]
    install ReportFreeMemorySubscription on [Network|*|NodeID()=48]

for [Network|*|NodeType()="Gateway"]
    install Service1Subscription on [Network|*|NodeID()=48 || HasSysCall(Temp)]
```

Listing 3.6: Scenario.wsp

Figure 3.9 shows the Loader and Simulator after the new application is installed. Before the new scenario was installed, the memory map was (4514, 4564, 2); and after the new scenario was installed, the memory map became (4406, 4462, 3).

The memory usage is:

\[
\text{Memory usage}_{\text{main scenario}} = \text{Free memory}_{\text{before}} - \text{Free memory}_{\text{after}}
\]

\[
= 4564 - 4462 = 102B
\]

(3.1)

In Sections 3.7, “Memory model evaluation”, we compare measured memory usages with the estimates from the models.
3.6 Memory analysis tool

A memory analysis tool was built out of the models we presented. It was written in C++, and it has about six hundred (600) lines-of-code. The tool takes a wbc file as an input, and gives the memory usage breakdown of the program (in the form of an output file). We demonstrate the use of the tool though an example.

3.6.1 Using the memory analysis tool: Example

Consider the OSAS program in Listing 3.7.

```
service ReadTemp($Handler, $limit)
  define
    count := 0
```
on event read when Temp() > $limit && count < 50 do
    count := count + 1;
    SendToSubscribers($Handler, count)

subscription ReadTempSubscription
to ReadTemp($Handler=print, $limit=400)
with (period=100ms, deadline=1s, send="Normal", exec="Normal")

for [Network]*|NodeID()=48]
    install ReadTemp

for [Network]*|NodeType()="Gateway"]
    install ReadTempSubscription on [Network]*|NodeID()=48]
.

Listing 3.7: Example program

The resulting wbc file is given as an input to the tool. Listing 3.8 shows the run; and Listing 3.9 gives the memory analysis results (The tool’s output file).

Listing 3.8: A run of the memory analysis tool
Chapter 3. Memory model

----------- Memory analysis of input.wbc --------------

Memory usage values are given in the form: "a Bytes | b Bytes | c Bytes | d Bytes | e Bytes" where a, b, c, d, and e are the Model 1, Model 2, Model 3, Model 4, and average memory usages respectively.

CBAs:
--------
address0: 16 Bytes | 22 Bytes | 16 Bytes | 22 Bytes | 19 Bytes
address1: 15 Bytes | 21 Bytes | 16 Bytes | 22 Bytes | 19 Bytes
CBAs total: 31 Bytes | 43 Bytes | 32 Bytes | 44 Bytes | 38 Bytes

Services
--------
ReadTemp
State: 24 Bytes | 30 Bytes | 24 Bytes | 30 Bytes | 27 Bytes
Event-generators:
read: 29 Bytes | 35 Bytes | 30 Bytes | 36 Bytes | 33 Bytes
Event-generators total: 29 Bytes | 35 Bytes | 30 Bytes | 36 Bytes | 33 Bytes
ReadTemp total: 53 Bytes | 65 Bytes | 54 Bytes | 66 Bytes | 60 Bytes

Services total: 53 Bytes | 65 Bytes | 54 Bytes | 66 Bytes | 60 Bytes

Subscriptions
--------
ReadTempSubscription: 28 Bytes | 34 Bytes | 28 Bytes | 34 Bytes | 31 Bytes
Subscriptions total: 28 Bytes | 34 Bytes | 28 Bytes | 34 Bytes | 31 Bytes

Program total: 112 Bytes | 142 Bytes | 114 Bytes | 144 Bytes | 129 Bytes

LISTING 3.9: Memory analysis results (tool output file)

From Listing 3.9 we observe that the example application has two CBAs, one service, and one subscription. The service in turn has a state and an EG. The tool gives five memory usage values according to Model 1, Model 2, Model 3, Model 4, and the average of the previous four, respectively. Note that the tool gives the memory usage of every part shown in Figure 3.4. It also gives the total memory usages at each stage. The detailed memory usage analysis makes sure that we can investigate the memory usage not only for the whole application, but all for its parts.
3.7 Memory model evaluation

The model was evaluated through 18 example applications, some of which taken from the WASP project. The result is shown in Figure 3.10.

Figure 3.10: Memory model evaluation

Figure 3.11 shows the over/under estimation of each model for each application. From the figure, we have have the order Model 3 → Model 1 → Model 2 → Model 4 in increasing order of over/under estimation. The average over/under estimation of each model is given in Figure 3.12 which confirms that Model 3, which considers only internal fragmentation, has the closest estimate with about 3% error. Models 2 and 4 give have high over-estimation because they consider external fragmentation overhead for every allocation although this happens very infrequently.
### Chapter 3. Memory model

#### Figure 3.11: Model 3 → Model 1 → Model 2 → Model 4 in increasing order of over/under estimation

#### Figure 3.12: Model 3 provides the closest estimate (3% error).
Chapter 4

Conclusions and further work

4.1 Summary and Conclusions

In this thesis, we designed ET and memory models for OSAS software components, and we built tools for them. These tools are used during OSAS application development to test for resource requirements. Testing applications for resource requirements is not only important for ensuring the correct functionality of the applications, but also to analyze the quality of service. We now briefly summarize the tools.

4.1.1 The ET analysis tool

This tool takes a wbc file as an input; and gives the ET breakdown of each execution unit and furthermore each program construct. It also gives the BCET and WCET of the execution units. With this knowledge, we can now correctly set subscription periods and tune applications for performance. The tool is written in C++, and it has five thousand (5000) lines-of-code.

4.1.2 The memory analysis tool

This tool a wbc file as an input and gives the memory usage breakdown of each execution unit. Since wireless nodes are memory constrained systems, it is important to test applications for memory requirements. For example, an application may want to collect data for some time and aggregate it. Collecting data over a substantial amount of time
leads to a considerable memory consumption, and the required amount of memory may not be available. Using the memory analysis tool, we can now see, without deploying, if the target node can fulfill the memory requirements of an application. The tool is written in C++, and it has about six hundred (600) lines-of-code.

4.1.3 Revisiting the problem statement

Section 1.3, “The problem statement”, gave the research questions that we need to answer in this thesis. In this section we revisit these research questions and answer them with respect to the contributions this thesis made.

4.1.3.1 static and dynamic parts of the resource models

The ET model has \#iterations parameter for while loop constructs as a dynamic part. The remaining parts of the ET model are derived statically. For the memory model, the OSAS memory layout, Figure 3.4, is the static part, and the internal and external fragmentation overheads are the dynamic parts.

4.1.3.2 Measuring and specifying the dependence of resource use on runtime parameters

The ET model gives the BCET and WCET of all execution units in a wbc file. The result has \#iterations for while loops as parameters. The \#iterations for while loops can be determined by simulating the application. The ET model we developed here expects the user to provide the \#iterations values.

4.1.3.3 Model integration and evaluation

Both the ET and memory models require the wbc file, which is an output of the compiler. Another possibility is to integrate the resource models to the compilation process.

Both the ET and memory models have been evaluated with test applications, some of which from the WASP project. The BCET and WCET as determined by the ET analysis tool were checked against the measured values. The WCET is larger than the
measured ET; and the BCET is lesser than the measured values. The memory model was able to predict the memory usage with only 3% error.

4.1.3.4 What to Use the models for

The ET model is used to set subscription periods and to conduct an acceptance test during development. The memory model is used to conduct an acceptance test for memory requirements.

4.2 Further work

4.2.1 Other resources

Other resources that are important to model for WSNs are energy and communication bandwidth. Modeling communication bandwidth in OSAS requires the proper handling of CBAs.

4.2.2 Taking CBAs into consideration

CBAs by definition can only be evaluated at the target node, and therefore are difficult to model/simulate. For example the CBA $[\text{Network}] * [\text{Temp}() > 40]$ is difficult to evaluate at development time as the temperature reading can only be made on the actual node after deployment. For resource modeling, it is important to know which applications get installed where without deployment. For that the CBAs have to be evaluated statically or using a feedback mechanism; and the challenge is on how to do this with a fair level of accuracy.

4.2.3 Determining the #iterations of while loops

#iterations for while loops is mostly determined from annotations to the source code[22]. The OSAS source code can similarly be annotated with #iterations of while loops; and this information can later be used to by the ET model. This task needs a compiler support.
4.2.4 Extending the ET analysis tool with nesting

The ET analysis tool can be extended so that it also works with applications having nested blocks (See Section 2.7.2). To assist this extension, the Process-wbc-File part of the tool (See Figure 2.28) already identifies all blocks and puts this information to separate output files named after the block types.
Bibliography


