TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Particle Swarm Optimization for Generating Input Data in Measurement Based Worst-Case Execution Time Analysis

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Science
an der Technischen Universität Wien,
Fakultät für Informatik

eingereicht von
Miljenko Jakovljević
Matrikelnummer: 0426673
E-Mail: miljenko.jakovljevic@gmail.com

Betreuer
Univ.-Ass. Dr. Raimund Kirner
Mitwirkung: Projektass. Dipl.-Ing. Michael Zolda
Institut für Technische Informatik 182

Wien, am 28.01.2011

Unterschrift des Studierenden

Thank you!

I would like to thank my mentors Raimund Kirner and Michael Zolda for their patience and help in making this work better.

Special thanks go to Michael for proofreading the manuscript, advice on the graphical display of data, and for the initial ‘clang hack‘.

This work is dedicated to my parents Miro and Gordana, for their love and support.

Contents

1 Introduction **17**

2 Worst Case Execution Time Analysis **19**
2.1 Problem Statement . . . 19
2.1.1 Frequency distribution of program execution times . . . 19
2.1.2 Often used terms for describing WCET bounds . . . 21
2.1.3 Decidability of WCET . . . 21
2.2 Factors Influencing WCET . . . 21
2.2.1 Program structure . . . 22
2.2.2 Program translation . . . 23
2.2.3 Scheduling of the program for execution . . . 23
2.2.4 Execution on the processor system . . . 24
2.2.5 Interactions with the environment . . . 24
2.2.6 Summary . . . 25
2.3 Concepts and Definitions . . . 25
2.3.1 Basic block . . . 25
2.3.2 Control flow graph . . . 25
2.3.3 Program segment . . . 26
2.3.4 Syntax tree . . . 26
2.3.5 Program execution path . . . 27
2.3.6 Real time system . . . 27
2.3.7 Causes of variance of program execution time . . . 28
2.4 Step 1: Program Source Code Analysis . . . 28
2.4.1 Programming language . . . 29
2.4.2 Control flow analysis . . . 29
2.4.3 Flow fact annotations . . . 30
2.5 Step 2: Timing Information Derivation . . . 31
2.5.1 Processor pipeline . . . 31
2.5.2 Processor cache memory . . . 33
2.5.3 Speculative execution . . . 34
2.5.4 Timing anomalies . . . 34

2.5.5 Effect of timing anomalies on WCET analysis 36
2.6 Step 3: WCET Bound Calculation 37
2.6.1 Path-based calculation 38
2.6.2 Tree-based calculation 38
2.6.3 IPET-based calculation 42
2.7 Types of WCET Analysis 46
2.7.1 Static WCET analysis 47
2.7.2 End-to-end measurement 48
2.7.3 Hybrid WCET analysis 49
2.8 Overview of WCET Tools and Related Work 51
2.8.1 Completely static WCET tools 51
2.8.2 Mostly static and hybrid WCET tools 55
2.8.3 Completely measurement based WCET tools 57
2.8.4 Related work 58

3 Particle Swarm Optimization 59
3.1 Introduction 59
3.2 The Continuous PSO Algorithm 60
3.2.1 The continuous search space 60
3.2.2 Definition of a PSO particle 60
3.2.3 Algorithm pseudocode 61
3.2.4 Particle initialization 61
3.2.5 Particle evaluation 63
3.2.6 Update of particle history best position 63
3.2.7 Update of particle group best position 64
3.2.8 Update of particle velocity 64
3.2.9 Update of particle position 66
3.2.10 Values for the cognitive and social constant 67
3.2.11 Limitations of the continuous algorithm 68
3.3 The Discrete Binary PSO Algorithm 68
3.3.1 The probability search space 69
3.3.2 The discrete binary particle 69
3.3.3 Algorithm pseudocode 70
3.3.4 Deriving of realized position from probability position 71
3.3.5 Decoding of realized position and calculation of fitness . . . 71
3.3.6 Update of probability velocity 72
3.3.7 Update of probability position 72
3.3.8 Maximum value of probability velocity 73
3.3.9 Statistic analysis of the update of probability velocity 73
3.4 Algorithm Modules 75
3.4.1 Topology of the Particle Swarm 75

3.4.2 Distance metric . . . 77
3.4.3 Neighborhood relation . . . 80
3.4.4 Effect of topology on performance . . . 81
3.4.5 Encoding of position coordinates . . . 83
3.5 Extensions of the PSO Algorithm . . . 84
3.5.1 Manipulation of particle velocity . . . 84
3.5.2 Spatial extension of particles . . . 86
3.6 Summary . . . 88

4 Adaptation of PSO for Estimating WCET 91
4.1 Services of the PSO WCET Framework . . . 91
4.1.1 Preparing the software under test . . . 92
4.1.2 The testing cycle . . . 93
4.1.3 Saving the results of optimization . . . 93
4.2 Software Under Test . . . 93
4.2.1 Main function signature . . . 94
4.2.2 Intermediate representation . . . 94
4.2.3 Instrumentation . . . 95
4.3 Particle Swarm Optimizer . . . 95
4.3.1 Algorithms . . . 95
4.3.2 Fitness functions . . . 95
4.3.3 Other modules of the optimizer . . . 100
4.4 Configuration . . . 101
4.5 Applications . . . 101
4.5.1 Search for the longest execution path . . . 101
4.5.2 Search for the WCET inducing execution path . . . 101
4.5.3 Exercising a specific execution path . . . 103
4.5.4 Exercising an execution path neighborhood . . . 104
4.5.5 Checking execution path feasibility . . . 104
4.6 Related Work in Software Testing . . . 105
4.6.1 Types of software testing . . . 105
4.6.2 Algorithms for generating input data . . . 106
4.6.3 Types of fitness functions . . . 106
4.7 Related Work in Measurement Based Timing Analysis . . . 107
4.8 User Manual . . . 108
4.9 Summary . . . 113

5 Experiments 115
5.1 Maximization of the WCET Estimate . . . 116
5.1.1 Experiment 1: Effect of topology, metric, and encoding . . . 116
5.1.2 Experiment 2: Effect of other PSO parameters . . . 118

5.1.3 Experiment 3: Maximum velocity and mean performance . . 125
5.2 Optimization of Rastrigin's Function 132
5.2.1 Experiment 1: Effect of topology, metric, and encoding . . . 133
5.2.2 Experiment 2: Effect of PSO learning factors 137
5.2.3 Experiment 3: Maximum velocity and mean iteration count 139
5.3 Summary 141

6 Conclusion 145

7 Bibliography 147

List of Abbreviations

BCET - Best-Case Execution Time

CFG - Control Flow Graph

CPSO - Continuous Particle Swarm Optimization

DPSO - Discrete Particle Swarm Optimization

IPET - Implicit Path Enumeration Technique

LLVM - Low Level Virtual Machine

MBTA - Measurement Based Timing Analysis

PSO - Particle Swarm Optimization

SUT - Software under Test

WCET - Worst-Case Execution Time

List of Figures

2.1 Relative Execution Time Frequencies of a Computer Program. . . . 20
2.2 Phases of Traditional Program Development and Life Cycle. 22
2.3 CFG of an Exemplary Program. 26
2.4 Syntax Tree of a *For* Loop. 27
2.5 CFG of an *If-Then-Else* Alternative 44
2.6 CFG of a *While* Loop. 45
2.7 Relationship Between the Number of Program Segments and the Total Number of Execution Paths in Segments 50

3.1 Change of the Particle Velocity Vector 65
3.2 Geometric Interpretation of Update of Particle Velocity and Position 67
3.3 Particle Swarm using a Star Topology 76
3.4 Particle Swarm with a Circle Topology 77
3.5 Particle Swarm with a Wheel Topology 78
3.6 Coordinate Based vs. Fitness Based Metric 82
3.7 Bouncing of Particles 88

4.1 Data-Flow Diagram of *pso_wcet* 92
4.2 Instrumented LLVM Code of SUT `bubble sort` and the Automatically Generated CFG 96

5.1 Influence of PSO Learning Factors on the WCET Estimate of `bubble sort` Obtained by CPSO. 119
5.2 Influence of PSO Learning Factors on the WCET Estimate of `bubble sort` Obtained by DPSO. 120
5.3 Influence of Maximum Particle Velocity on the WCET Estimate of `bubble sort` Obtained by CPSO. 121
5.4 Influence of Maximum Particle Velocity on the WCET Estimate of `bubble sort` obtained by DPSO 122
5.5 Influence of Particle Count on the WCET Estimate of `bubble sort` Obtained by CPSO. 124

5.6 Influence of Particle Count on the WCET Estimate of bubble sort Obtained by DPSO. 125
5.7 Influence of Iteration Count on the WCET Estimate of bubble sort Obtained by CPSO. 126
5.8 Influence of Iteration Count on the WCET Estimate of bubble sort Obtained by DPSO. 127
5.9 Annotated CFG of SUT bubble sort with Synthetic, Worst-Case Execution Times of Basic Blocks. 128
5.10 Influence of Maximum Particle Velocity on the Mean Performance of CPSO and DPSO in Estimating WCET 131
5.11 Graph of the Rastrigin's function. 134
5.12 Influence of Learning Factors on the Optimization of Rastrigin's function by CPSO. 138
5.13 Influence of Learning Factors on the Optimization of Rastrigin's function by DPSO. 139
5.14 Influence of Maximum Particle Velocity on the Mean Performance of CPSO and DPSO in Optimizing the Rastrigin's function. 142

List of Tables

3.1 Different Cases of Probability Velocity Update for DPSO. 74
3.2 Example of Gray Encoding 84

4.1 Parameters of PSO Configuration 102
4.2 Parameters of SUT Configuration 102
4.3 Comparison of Related Work in MBTA. 108

5.1 Default Configuration of CPSO and DPSO. 115
5.2 Configuration of the Optimization Problem `bubble sort` 116
5.3 General Configuration of PSO for Experiments on `bubble sort`. . . 116
5.4 Effect of Topology, Metric, and Encoding on WCET Estimates of `bubble sort` by DPSO, CPSO, and *random search*. 117
5.5 Special Configuration of PSO for Experiment 2 on `bubble sort`. . . 118
5.6 Comparison of WCET Estimates Obtained by Tree-Based Formulas, Analytic Means, and *pso_wcet*. 129
5.7 Selection of Learning Factors for Experiment 3 on `bubble sort` . . 130
5.8 Configuration of Experiment 3 on `bubble sort`. 130
5.9 Configuration of the Optimization Problem `Rastrigin`. 133
5.10 General Configuration of PSO for Experiments on `Rastrigin`. . . . 133
5.11 Default Configuration of PSO for Experiments on `Rastrigin`. . . . 135
5.12 Effect of Topology, Metric, and Encoding on the Optimization of `Rastrigin's` Function by DPSO, CPSO, and *random search* 136
5.13 Configuration of PSO for Experiments 2 and 3 on `Rastrigin` 137
5.14 Selection of Learning Factors for Experiment 3 on `Rastrigin` . . . 140
5.15 Special Configuration of PSO for Experiment 3 on `Rastrigin`. . . . 141

Chapter 1

Introduction

The objective of *worst-case execution time analysis* - also known as WCET analysis - is to find the longest possible running time of a certain software component on a given hardware platform. The running time depends on the component itself, the input data vectors, and the execution environment.

Knowledge of the WCET is especially important in *hard real-time systems*. In such systems all tasks have to finish before a predefined deadline. Failing to do so may result in catastrophic system failure. The maximum execution time of each task has to be known beforehand in order to choose an appropriate schedule and to guarantee system safety.

The methods for estimating WCET can be divided into *static* and *dynamic methods*. Static methods analyze the source code and model the target architecture that the software component is executed on. By combining these two, the worst case execution time of a software component can be estimated.

Dynamic methods obtain the WCET empirically. A software component is executed with different input vectors, and its execution time on a given platform is measured. Some tools for WCET estimation use a combination of both static methods and measurements; they are usually classified as *hybrids*.

In this bachelor thesis, a hybrid method with elements of both dynamic and static WCET analysis is implemented. Currently, the method uses synthetic values for the execution time cost of single instructions. It exercises the software-under-test with different input data generated in an attempt to induce the longest execution time. The input data are generated by *particle swarm optimization*. Besides the maximization of execution path length, several fitness functions for *structural software testing* are implemented as well.

The method can be used as a stand-alone tool, in which case it provides a synthetic, lower WCET bound. The lower bound can be useful in recognizing problems with the temporal specification early on in the development cycle of a real-time application. The method could also be used to provide services to a broader framework for WCET analysis, such as generating input data and exercising execution paths.

Chapter 2

Worst Case Execution Time Analysis

2.1 Problem Statement

Execution time analysis of computer programs deals with program execution time, in particular with finding bounds for the best case (BCET) and the worst case (WCET) execution times. Estimates of the WCET are usually the more interesting of the two execution time extremes, since the knowledge of WCET enables validation of timeliness properties of real time systems.

2.1.1 Frequency distribution of program execution times

The execution time of a computer program is determined by the input data supplied to it and by the initial state of the processor system on which it executes. Different combinations of input data and initial processor states can induce the same program execution times. For each observed program execution time t_j, there exists a corresponding set S_{t_j} of pairs of input data and initial processor states, whose elements induce the program execution time t_j.

$$f(t_j) = \frac{|S_{t_j}|}{\sum_{i=1}^{n} |S_{t_i}|} \tag{2.1}$$

In Equation 2.1, $f(t_j)$ is the ratio of the number of sample input-data/initial-processor-state pairs, inducing the execution time t_j, and the total number of performed execution-time measurements; it corresponds to the relative frequency of program execution time t_j. Relative execution time frequencies of an exemplative program are illustrated in Figure 2.1. In this figure the WCET value has been obtained as a result of exhaustive execution time measurements over the space of

input data and initial processor states. Without some restrictions this is generally considered to be unfeasible. For example, total coverage of a single *int* parameter in C would require the testing of 2^{32} different numeric values. Therefore, Figure 2.1 is an idealization; in practice it may be difficult to obtain for some program on a given platform.

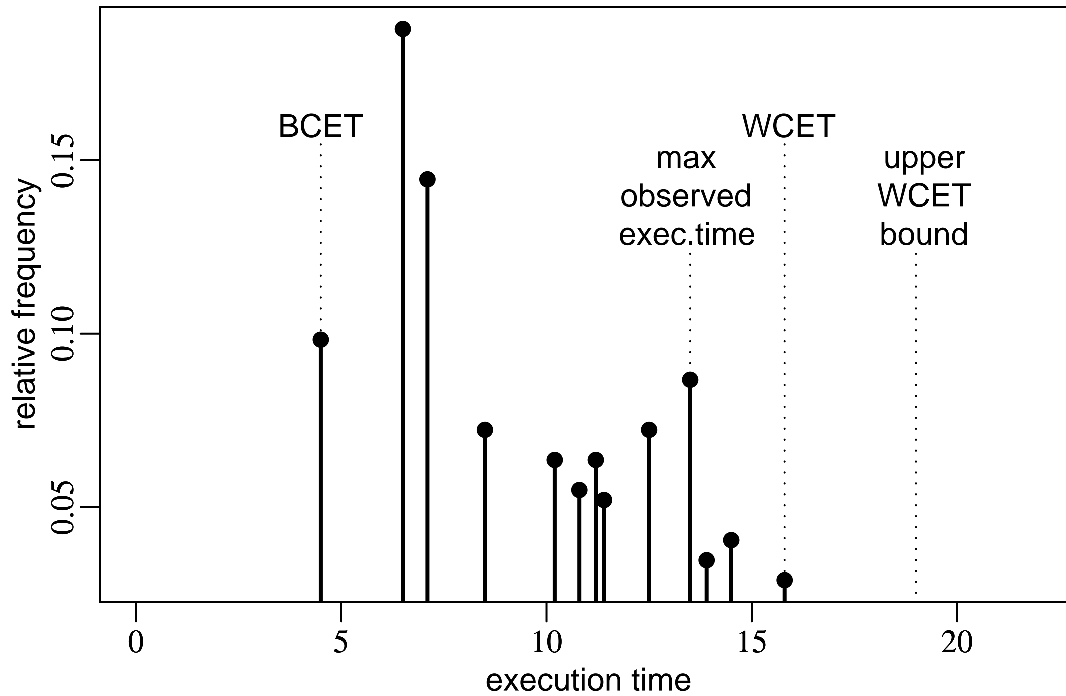

Figure 2.1: Relative execution time frequencies of some computer program as would have been obtained by exhaustive testing. As a result of exhaustiveness, one would have found the maximum execution time, i.e. the program *WCET*. If exhaustiveness cannot be guaranteed, then one can only speak about the *maximum observed execution time*. The difficulty of exact WCET derivation is depicted in this figure by assigning the lowest relative frequency to the worst-case execution time. An alternative to measurement based WCET derivation are static methods. If implemented correctly these methods can provide an *upper WCET bound*, an overestimation of worst-case execution time.

2.1.2 Often used terms for describing WCET bounds

There are certain characteristic terms used when dealing with the properties of WCET estimates.

1. A *safe WCET estimate* is greater than or equal to the real WCET of a program.

$$(WCET_{est} \geq WCET_{real}) \tag{2.2}$$

2. The terms *precision* and *tightness* are similar; they state that the WCET estimate should be close to the real WCET of a program.

3. Corresponding to precision and tightness are the terms *pessimism* and *overestimation*. They quantify the degree whereby the upper estimate is higher than the real WCET.

Sometimes the term WCET is interchangeably used in literature. At one time it is used to denote the exact WCET of a program, and at other times it means the estimated program execution time which bounds the real program WCET. Here, the term WCET will be used to denote the real worst-case execution time, although the exact value may be unknown or difficult to obtain for most programs.

2.1.3 Decidability of WCET

It was long assumed, that the problem of finding the exact WCET and BCET is undecidable in general, since, as the argument went, finding it would be at least as difficult as solving the Turing's *Halting problem*. Only a safe overestimate of the WCET could be feasibly obtained.

In contrast, a recent finding [1] showed that a class of programming languages, among these C and subsets of Java, are equivalent to push down automata which are decidable. It follows, that the exact WCET can be obtained for programs written in those languages. Furthermore, it is also reported that the property of decidability is not only influenced by the specification of the programming language, but also by its implementation details for a concrete hardware platform.

2.2 Factors Influencing WCET

The factors influencing the program execution time can be viewed in a top down approach, where each level corresponds to a phase of the traditional program development and life cycle. This is graphically depicted in Figure 2.2. At the top level there is an idea of computation that is formalized by the programmer

into the form of a computer program. At the middle level are the translation of a computer program into the assembly and ultimately binary code, and its scheduling for execution. The execution of a program as instructions on a processor system belongs to the lowest level that is considered in WCET analysis. Below that level starts the domain of physical effects inside the processor system, which belong to the study of microprocessor engineering.

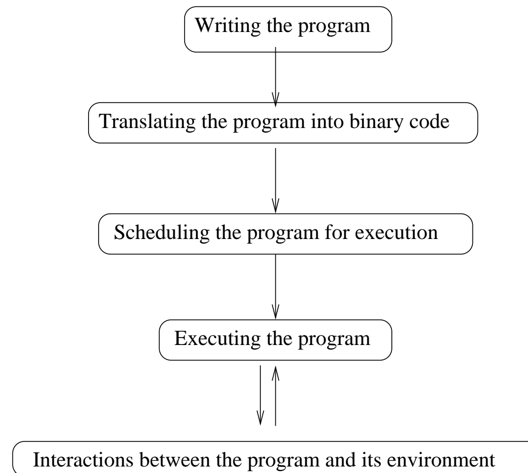

Figure 2.2: Phases of traditional program development and life cycle.

2.2.1 Program structure

The main elements of a computer program written using a high-level, structured, imperative programming language such as C are variables, operators, assignments, and function and system calls.

Elements linking the above into a logical whole are called *flow-of-control* statements. They determine how often and upon what condition parts of program code will be executed. Examples of flow-of-control statements are: *for*, *while*, and *do* loops, *if-then-else*, and *switch-case* alternatives. Despite their multitude, all of the high-level flow-of-control constructs rely on a simple conditional-branch instruction for their respective implementations in assembly.

A single flow-of-control statement influences the execution time of a computer program in a twofold manner. The first aspect is the time needed to evaluate each statement itself, which consists of the logical condition inside and the jump to a different part of program code. The second aspect is the time spent executing the code to which execution was directed by the flow-of-control statement.

Flow-of-control statements, thus, determine the set of possible code sequences that can be executed for a given program. Which one of those sequences is executed at runtime depends on the conditions inside the flow-of-control statements. Three kinds of flow-of-control conditions can be distinguished.

1. First, there are those conditions in the flow-of-control statements that always evaluate to the same value, since they are either a truism in themselves or use some statically predefined value as a variable.

2. The second possibility are control-flow conditions whose evaluation is based on a random value and whose outcome cannot be determined. This is a special case and a minority among control-flow conditions used in practice.

3. The third and most common case is that the control-flow condition is evaluated based on the variables that directly or indirectly depend on the program's input data vectors. In this case the exact sequence of executed code is a function of: (a) the program control-flow structure defined during program development and compilation, and (b) the input data provided at runtime.

2.2.2 Program translation

A program is translated into assembly code by the compiler. Since the assembly representation of the program corresponds one to one with the machine code executed on a processor system, static code analysis for WCET is usually done on this level.

The disadvantage of low-level analysis is that high-level, flow-of-control constraints can be lost due to optimizations performed during compilation. State of the art WCET tools combine high with low-level code analysis; structural and functional flow constraints are obtained on the high and intermediate level, while time constraints of single *basic blocks* are obtained on the low level.

2.2.3 Scheduling of the program for execution

If a program is not running alone on a dedicated platform, e.g. a microcontroller, but concurrently with other programs, then it needs to be scheduled for execution, usually by an operating system scheduler. In that context programs are usually refferred to as tasks. If tasks can be preempted during their execution this can have a twofold impact on WCET.

First there is the more or less constant overhead of task switching that is taken care of by the appropriate scheduler code. Analysis of that code can give the WCET of the overhead. The second, more difficult impact of preemption on WCET analysis is the possible emptying of the cache content of some task upon

it being preempted by another task. Once the original task continues execution, it may start with an empty cache. This can pose a challenge to *static WCET analysis*, since it may warrant a pessimistic assumption that the task code always operates with an empty cache, i.e., that memory accesses always cause a cache miss.

In WCET analysis the topic of scheduling interferences is usually abstracted away; it is assumed that a program is running on a dedicated processor or that tasks cannot be preempted.

2.2.4 Execution on the processor system

After the program executable has been copied into memory and the processor instruction pointer has been set to program start, the execution can begin. Ultimately, it is the effects in the processor system that determine the execution time and correspondingly the WCET of a computer program.

The processor core is one part of the processor system. The remaining parts: instruction cache, data caches, and main memory also have to be included in the WCET model of the execution environment. External storage and other peripheral components are abstracted away in WCET analysis, and effects such as spilling the main memory to the hard disk, i.e. swapping, are not considered.

On many processors systems, the execution time of a program can be affected by the initial state of the processor at the start of execution. For example, a processor which is in power save mode on execution start, might need a few more cycles than a processor which begins execution from the ready state.

2.2.5 Interactions with the environment

A program interacts with its environment by accepting input data and providing output.

Based on when the new input data are received by the program, one can distinguish reactive and transformative systems. Reactive systems keep on running continuously, interacting with the environment. Transformative systems are characterized by their one shot behavior: the input data is taken, processed, and a result is produced as output. After that, a transformative system terminates. A transformative system is usually implemented as a task to be executed periodically. The WCET analysis mostly deals with such systems, since they are easier to analyze and can serve as building blocks of more complex systems.

Different input data exercise different program control-flow paths and ultimately cause different instruction sequences to be executed. It is impossible to analyze most programs for all possible input data vectors, as the input space is huge. As opposed to *static analysis*, one strategy to overcome this complexity is

to aim for good coverage of program execution paths. This is achieved by exercising the program with a limited set of representative input data vectors. For this purpose, heuristic techniques such as genetic programming have been used in the past.

2.2.6 Summary

The execution time of a program is influenced by static and dynamic factors. Static factors are the program structure itself and the architecture of the processor system: processor, cache, and main memory. They are invariant for some program running on a certain processor system. The dynamic factors are the program input data and the initial state of the processor system. They may change at run-time and are responsible for making WCET analysis difficult.

2.3 Concepts and Definitions

Since the structure of the computer program has a big influence on the WCET of the program, static program analysis is used in almost all WCET methods. Before going into the details of different kinds of WCET methods (Section 2.4,2.5,2.6), it is appropriate to give the definitions of the concepts that will be used in later sections. The following concepts dealing with program structure are defined: *basic block*, *segment*, *control-flow-graph*, *execution path*, and the *syntax-tree* of a computer program. All the here numbered concepts play a role in almost any modern WCET analysis framework. Furthermore, since guaranteeing the temporal correctness of real-time systems is the raison d'être for the study of program worst-case execution time, a definition of a *real-time system* is given here as well.

2.3.1 Basic block

A *basic block* is a maximal sequence of one or more instructions with a single entry point, where the flow of control can enter, and a single exit point, where the flow of control can exit the sequence. Consequently, control transfer instructions (jump, call, and return instructions) can only be present at the end of basic blocks.

2.3.2 Control flow graph

A *control flow graph* (CFG) describes the flow-of-control for each function in the program. The nodes of the graph designate the basic blocks and the directed edges designate the possible transfer of control between the basic blocks. An exemplary CFG is shown in Figure 2.3.

Figure 2.3: CFG of an exemplary program.

2.3.3 Program segment

A program *segment* is a subgraph of the program CFG. It is defined by a starting basic block B_1, an ending basic block B_n, and a set of intermediate basic blocks $S = \{B_2..B_{n-1}\}$. Each execution path spawned by the segment starts with B_1, ends with B_n and contains a sequence of basic blocks from S inbetween.

2.3.4 Syntax tree

A program *syntax tree* is a tree, where nodes denote the structural elements of programs in a high-level language. For the purposes of WCET analysis each intermediate node denotes a language construct occurring in the source code, while leaves represent basic blocks. Syntax tree nodes can be annotated with additional information describing the possible behavior of the program. Such information can be provided by the user or it can be obtained automatically by data analysis of the program code. An example of a program syntax tree is shown in Figure 2.4.

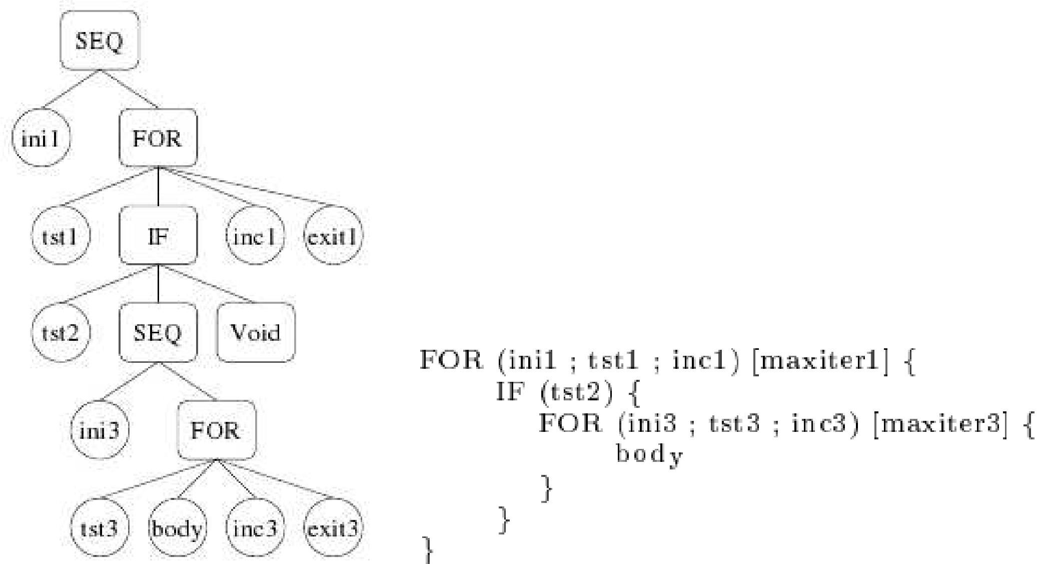

Figure 2.4: Syntax tree of a *for* loop[2].

2.3.5 Program execution path

A program *execution path* is a sequence of basic blocks, which denotes the path taken through the program CFG. Possible paths depend on the control transfer constructs in program code and on the input data. In case of error-free execution, each path starts at the program entry and ends at the program exit point. Paths which are structurally possible but cannot occur at run-time because of program semantics are called unfeasible.

2.3.6 Real time system

A *real time system* is composed of tasks which need to be finished inside certain time intervals. Each task can be a program running as a separate process or a function or procedure called from the main function of a single process.

A real time system may be periodic or aperiodic, depending on whether the tasks are executed repeatedly or only once. In both cases each task has a deadline until it has to finish execution.

In soft real time systems, meeting the deadline of a task is desirable but not mandatory. It makes sense to schedule and complete the task even after the deadline has passed. On the contrary, in hard real time systems, a missed deadline can cause a system failure.

It is the task of scheduling analysis to determine a task execution schedule, so

that all tasks finish inside their given deadlines. Hard real time systems employ offline scheduling analysis which produces a static plan according to which tasks are dispatched at run-time. The main benefit of static scheduling is its predictability. Thus, it can be guaranteed that no task will miss its corresponding deadline.

Traditional literature about task scheduling algorithms abstracts away from the problem of real task execution times in their analysis, by supposing them as constant and known beforehand [3, p.461]. This is acceptable for a theoretical model, but not for a system that has to obey time constraints in practice. In order for a theoretical scheduling model to be safely applied to a hard real time system, it has to be extended by worst case execution time analysis of tasks' program code.

2.3.7 Causes of variance of program execution time

The variance of the execution time of computer programs, observable in the execution time frequency diagram, Figure 2.1, can be attributed to two principal causes.

The first cause is that different input data can exercise different program execution paths. Different execution paths contain different sequences of instructions which is the cause of the variance in execution time.

The second cause of variance of program execution time is attributed to the variance of execution times of single processor instructions. Hence, the same execution path, i.e., the same sequence of processor instruction can take different amounts of time to execute on the same processor system. This kind of variance can be *data-dependant* or *history-dependant* [4].

Instructions with data-dependent execution times show different temporal behavior based on the input data. Multiplication and division are examples of operations that can take a data-dependent amount of time for completion. However, the property depends on the exact hardware implementation of the respective instructions.

History dependent execution times of processor instructions are the result of features that are intended to enhance the average-case performance. Examples in modern processors are: data caches, pipelines, and branch prediction. The cost of these enhancements is reduced worst case performance and reduced WCET predictability [5].

2.4 Step 1: Program Source Code Analysis

Program source code analysis is the first step of determining the worst-case execution time of a computer program.

In order to analyze programs for WCET, static and to a lesser degree dynamic WCET tools first have to analyze the program source code in order to build a model of the program. The simplest model is the program CFG. More complex models exist as well; their requirements depend on the type of WCET analysis (static or dynamic), the level of detail, and the amount of information the tool needs in order to perform the analysis. The complexity of the program model, especially for static WCET tools, rises with the level of detail of the performed analysis, with the accuracy of the WCET prediction, as well as with the complexity of the hardware platform, for which the analysis is performed.

2.4.1 Programming language

Many imperative programming languages include features that make *execution-time analysis* difficult or infeasible. Examples of such features are unbounded recursion, functions pointers, break, and goto statements. These features have to be disabled in order to perform static WCET analysis. Unbounded recursion results in a set of program paths that is potentially infinite. Function pointers complicate static identification of function call targets which is needed for the analysis of function calling contexts. Break and goto statements should not be allowed to cause breaks in instruction flow, as they can potentially invalidate the structure of loops and conditional program constructs [2]. Therefore, only a limited subset of conventional programming languages is analyzable for WCET in practice. Note that similar restrictions exist for the safe use of the C programming language in safety-critical embedded applications. For the automotive industry these have been specified by the MISRA consortium [6].

2.4.2 Control flow analysis

The analysis of flow-of-control transfer instructions gives information about the *control flow paths* (CFP) of a program. These describe the set of possible execution paths with applied execution constraints. For example, the constraints can be ranges for the values of input parameters.

The CFP are equivalent to a set of possible execution traces of a program. An execution trace can be obtained by instrumenting the basic blocks of program code with print statements or other kinds of instrumentation code.

A program which has unbounded loops has an unbounded set of CFP. Therefore, it is necessary to bound that set by the use of *flow facts*. The flow facts give hints about the possible CFP of a program. They can be obtained either implicitly by analyzing the program source, or they can be specified by the user.

2.4.3 Flow fact annotations

Flow fact annotation involves manually specifying *loop bounds* and *unfeasible paths* in the program code, or communicating them directly to the WCET analysis tool. This process can take as much as seven weeks of time for a medium sized industrial application [7]. Manual flow fact annotations are especially error prone for specifying unfeasible program paths, which are necessary for a tight WCET estimate. Without this information many unfeasible program paths would qualify for the program WCET. Furthermore, using the manual flow fact annotations requires that the developer has to update his flow facts for each functional change he makes in the program code, thus creating a further potential for errors.

2.4.3.1 Loop bounds

In spite of disadvantages, a certain minimum of flow fact annotations seems to be necessary, because of the limits of computability. The WCET problem without any flow fact information can be transformed to the well known *halting problem* which is provably undecidable [8]. The minimum of flow facts includes at least *loop bound annotations*. An example of a complete programming language with incorporated flow-fact annotations is *wcetC*, as described in [9]. It integrates the WCET analysis with the compiler, which is beneficial for analyzing the effects of compiler code optimizations upon the flow facts.

There have been several attempts to derive loop bounds automatically. Known approaches include *pattern matching*, *data-flow analysis*, and *abstract execution*. For loops with one loop test and a single increment, pattern matching provides tight and safe bounds. This method is limited to very simple non-nested loops. An improvement over pattern matching is the technique of data-flow analysis. It can bound loops with one loop test and more than one increment [10].

A further improvement of the loop bounds can be achieved by abstract execution, a method based on *abstract interpretation*. The idea is to extract properties of run-time behavior of the program by interpreting it using abstractions of values instead of concrete values. The program is executed over an abstract domain with abstract values for variables and abstract versions of the language operators. The abstract domain can, for example, be the domain of intervals. Each expression then evaluates to an interval rather than to a specific value, and each assignment will calculate a new interval from the current intervals held by the variables on the right hand side. A loop bound can be determined in this manner by iterating the abstract versions of the loop until the loop test returns false. The achieved number of loop iterations is then the worst case, i.e. the loop's upper bound [11].

After the upper *loop bound* has been obtained, the CFG of the loop can be analyzed, and out of it, the worst-case path of a single loop iteration can be found

by further flow analysis. If the estimated execution time of the loop body is t, and the upper loop bound is lb, then, using this approach, the estimated WCET of the loop would be calculated as $lb \cdot t$ [12].

2.5 Step 2: Timing Information Derivation

The gathering of timing information is the second step of WCET analysis. It concerns itself with low-level hardware effects upon the execution time of a program. This analysis can be divided into global and local low-level timing analysis, based on the hardware features it considers.

Global low-level timing analysis deals with those processor features which cause timing effects across the whole program. Among these are: *instruction caches*, *data caches*, *branch predictors*, and *translation lookaside buffers*. Global low-level analysis does not generate actual execution times, but rather, it only determines how certain hardware features might affect the program execution time. The results of the global low-level analysis are passed over to the local low-level analysis as execution facts.

With the help of execution facts, local low-level analysis can handle processor timing effects that depend on a single instruction and its immediate neighbors. Examples of local effects are overlapping of instructions in a processor pipeline, instruction alignment effects, and the effects of memory banks with different speeds (cache, main memory, ROM) that are common in real-time embedded hardware [13].

2.5.1 Processor pipeline

Pipelines are a feature found in virtually all desktop/server processors and many embedded processors. They offer a considerable speedup of instruction execution and lead to shorter clock cycles. Basically, the execution of an instruction is subdivided into stages. After one instruction completes a single stage of the pipeline, the following instruction may enter it, while the original instruction continues on to the next stage.

A typical pipeline of the *reduced instruction set computing* (RISC) architecture has the following stages:

Fetch stage Read instruction from memory.

Decode stage Read registers and decode instruction.

Execute stage Execute instruction or calculate a memory address.

Data stage Access operands in data memory.

Write stage Write the results into registers.

Modern processors have deeper pipelines where each stage is subdivided into smaller substages; for example, the Pentium 4 pipeline is 20 levels deep. The number of pipeline stages corresponds to the number of cycles that each instruction is in execution. More pipeline stages enable more instructions to be in execution simultaneously, and they induce shorter clock cycles. Empty pipeline stages during execution have a detrimental effect on performance and should be avoided.

Sometimes one instruction may depend on the results of another instruction and cannot enter the pipeline until the other instruction has completely finished execution. An example of such a scenario is the instruction sequence in Listing 2.1.

Listing 2.1: A sequence of instructions with mutual data dependencies awaiting scheduling.

```
ld r1, x;
add r1, y;
push r1;
mul r2,r3;
```

The `add` instruction has to wait for the `ld` instruction to finish, so that the register `r1` contains the needed value *x*. The `push` instruction also has to wait for the `add` instruction to finish before it can push the value of the register `r1` to the stack. Independent of these instructions is the `mul` instruction, operating on registers `r2` and `r3`. To prevent that free pipeline stages in the processor are wasted, instructions in this example should be reordered. The `mul` instruction should be executed not at the end, but while the `add` and `store` instructions are waiting for the `ld` instruction to finish.

Data dependencies between instructions can be identified, and instructions can be reordered during compilation. However, stalls in the availability of instruction operands during execution and conflicts in the use of processor *execution units* between the same type of instructions may bring about the reordering of instructions at run-time by the processor *pipeline scheduler*. Such processors, capable of out-of-order execution, may be difficult to analyze for WCET, since their instruction scheduling algorithms operate on a best effort basis. The exact execution order of instructions in those processors may be unpredictable and may complicate static WCET analysis, since it can not be known beforehand how many stalls will be caused in the pipeline for a certain instruction sequence.

A novel approach to make the WCET analysis of processor pipelines easier is to disable the dynamic scheduling unit of the processor, responsible for the run-time reordering of instructions. This can be achieved by injecting non-functional instructions into the program code. According to [14] there are at least two ap-

proaches to this. The first approach is to disable the *prefetch window* of the pipeline scheduler by periodically inserting *nop* instructions; the the second approach is to inject instructions that create artificial dependencies after each original instruction with variable execution time. These measures can remove any freedom of the processor scheduler to reorder successive instructions, i.e., to use a different instruction ordering from the one produced by the compiler.

2.5.2 Processor cache memory

Another feature of modern processors are memory *caches*. Historically, the rate of increase in processor speed has been higher than the corresponding increase in the speed of *main memory* (RAM). Thus, there now exists a gap between the computation speeds of processors and main memory. Overall execution time of instruction (especially for *load* and *store* instructions) is dominated by accesses to main memory. Faster processors would bring little computational power in return, if some mechanism to bridge this gap was not found.

Currently, computer systems implement a *memory hierarchy* consisting of the cache, the main memory, and the magnetic disk. The cache is the fastest memory, and it positioned nearest to the processor. Commonly, the cache is separated into the data cache and instruction cache. Both are expensive, and only limited storage space is available. A common technology for caches is SRAM.

Accesses to main memory can be slower by a factor of hundred than accesses to cache memory. At the same time main memory in the DRAM technology is fifty times cheaper than SRAM cache memory. Magnetic disk storage is the cheapest and the slowest technology for computer memory with access times of about 10^5 times longer than corresponding RAM accesses. Typically the bit cost of magnetic disks is 200 times less than the bit cost of RAM memory [15, p.469].

The speed gap is currently bridged by integrating the cache memory and the *processor core* in the same unit. As cache is more expensive, it cannot completely replace than conventional RAM memory. However, the principle of locality enables comparatively smaller caches to act as a buffer between the processor and the main memory. By temporal locality it is assumed that if a certain memory block is referenced at the current moment, it will also tend to be referenced again at a later time near to now. Spatial locality assumes that memory blocks whose addresses are near to each other will tend to be referenced more often than memory blocks with distant addresses. Based on these two principles data which is assumed to be needed in the near future is stored in the processor cache.

A *cache miss* occurs when some data that are needed by the processor during execution are not found in the cache. It can be both an instruction cache miss or a data cache miss. Upon a miss, data are read from the main memory and inserted into the cache. Depending on which old data block is replaced by the new block,

different cache replacement policies are distinguished: *round robin*, *least recently used* (LRU), *most recently used* (MRU), etc. The objective of these policies is to minimize cache misses and to maximize cache hits.

2.5.2.1 Impact of memory hierarchies on program execution time

The behavior of computer memory hierarchies has great impact on the performance of computer programs.

This used to be ignored for some time in traditional algorithm analysis. For example, traditional algorithm analysis of the *radix sort* algorithm states that it is an algorithm with linear execution time $O(n)$, which is better then the $O(n\, log\, n)$ execution time of *quicksort* [16, 15, p.173, p.508].

However, experimental assessment of both algorithms shows that the basic radix sort implementation causes more cache misses per item sorted than the quicksort implementation. The effect of cache misses is such that in practice quicksort has faster execution time than the basic radix sort, although radix sort is algorithmically more efficient, i.e., it needs fewer operations than quicksort.

2.5.3 Speculative execution

Branch prediction is an often used speed-up feature in modern processor architectures, where the processor speculation unit tries to determine, ahead of time, which program path will be taken. It fetches the instructions needed for the execution of the assumed program path and stores them in the instruction cache memory. When program execution reaches the branch condition, the processor realizes whether the outcome of speculation was correct or not. If it was correct, the speculative results are written to registers and memory and the instructions are completed. If the speculation was incorrect, then the processor flushes the speculation buffers and fetches the correct instruction sequence. Incorrect speculation causes an execution time delay, since in that case the right instructions first need to be fetched from the main memory, written to the cache, and finally inserted into the processor pipeline [15, p.434].

Speculation in modern processors is correct 90% of the time and, thus, offers a considerable increase in the average-case performance [17, p.270]. The downside of speculation is that the remaining 10% of incorrect speculation predictions can cause timing anomalies.

2.5.4 Timing anomalies

A *timing anomaly* occurs when a local increase of the execution time of one instruction causes an even greater increase or an overall decrease of the global execution

time in the future [18]. Likewise, a local decrease in the execution time of one instruction which causes an even greater decrease or a net increase of the global execution time in the future also creates an anomaly.

Processor features that cause timing anomalies as a result of variable execution times of single instructions are: dynamic scheduling into the pipeline, speculative execution (branch prediction), and some cache replacement policies [19]. Usually cache phenomena (hits or misses) and input data are responsible for the variance of the execution time of single instructions [18].

2.5.4.1 Scheduling anomalies

A necessary condition for scheduling based timing anomalies is the existence of *out-of-order resources* in the processor system [18]. Out-of-order resource can be multiple out-of-order execution units in the execute stage of the pipeline. They can process instructions in parallel as long as there are no data dependencies between the instructions.

Most execution units provide reservation stations, which serve as a queue for instructions. Once the execution unit is available it receives the next instruction that does not have any unsolved data dependencies from the queue. The previous instruction schedule and the execution time of instructions in other execution units determines the state of data dependencies and which instruction will be chosen for processing. Thus, if there are n instructions and each instruction has k possible execution times, then there are k^n possible execution schedules for a certain instruction sequence in the worst case [18]. Some of these schedules can be more compact than others based on the amount of parallel processing achieved by the execution units.

The worst-case schedule is the one in which no or little parallelism is achieved due to the unoptimal solving of instruction data dependencies. For WCET analysis, this instruction schedule is of most interest. However, since instructions can have variable execution time, it cannot of guaranteed that a certain execution schedule will result from a certain instruction sequence.

2.5.4.2 Speculation anomalies

Speculation anomalies are caused when the *branch prediction unit* in the processor wrongly speculates about the outcome of a program conditional branch and prefetches wrong instructions into the instruction cache. The misprediction can, however, be prevented by a cache-miss caused by instructions prior to the condition branch. Upon the cache miss, the instruction in question may use enough processor cycles so that there are not enough enough cycles left for the prefetching of the speculative instructions. A cache miss in the present may thus prevent a

more expensive branch misprediction in the future [19].

2.5.4.3 Cache anomalies

Cache replacement policies such as the *round-robin* policy can also cause timing anomalies. The reason for this is that the choice of a memory block to be over-written on *cache miss* may have beneficial influence on the execution time of the program in the future [20]. A cache miss of a single program instruction and the resulting cache-block replacement may prevent a series of future cache misses.

2.5.5 Effect of timing anomalies on WCET analysis

Processor timing anomalies invalidate two assumptions, which had previously been assumed in static WCET analysis. The first assumption was that the worst case state of the processor, as an initial state, will always leads to the WCET of the executed program. However, in dynamically scheduled processors this need not be true, since in the presence of scheduling anomalies a delay of one instruction can result in a more compact overall instruction schedule. The second assumption was that greedily taking the local WCET of single instructions and combining them together would always lead to the global WCET. [19] This may not be true for example in processors with branch prediction, since in some cases a local WCET of a single instruction (upon a cache miss) may prevent a more expensive branch misprediction.

2.5.5.1 Effect on measurement based WCET analysis

Timing anomalies pose a challenge for both static and measurement-based WCET analysis tools. Since in the presence of timing anomalies no processor state can be assumed apriori to be the worst case initial state, measurement-based tools need to conduct measurements starting from all possible processor states. Because of state explosion this approach is infeasible, and measurement based tools only use a subset of possible initial states in practice. Thus, classic measurement based tools are not safe for analyzing processors with timing anomalies [20].

2.5.5.2 Effect on static WCET analysis

For static WCET tools the challenge is to construct the model of a processor. The physical processor itself is deterministic; a successor state is always known from the known predecessor state, program code, and input data. Static WCET tools handle the large number of possible processor states by grouping them together using abstractions. Abstractions enable static WCET tools to be exhaustive in their analysis, but the price for it is the loss of precision. The loss of precision

leads to indeterminism of the processor model, which means that it is not always possible to conclude about the next state of processor execution from the known previous state, the program code, and the input data. After some known state there can be more than one possible successor states and in that case it is unknown which of them will really occur. Hence it becomes necessary to introduce the concept of an unknown state with high pessimism in the processor model. A local worst-case state was used often for the unknown state, but such an approach can no longer provide a safe WCET bound for modern processor architectures that exhibit timing anomalies [20].

2.5.5.3 Avoidance of timing anomalies

Timing anomalies can be completely eliminated by enforcing the in-order use of processor resources [18]. However, hardware support for disabling the dynamic instruction scheduler is currently lacking in most modern processors.

The processor instruction scheduler can also be disabled by a software-only approach. This includes inserting special non-functional instructions into the program code with the aim of creating data dependencies that guarantee the in-order use of processor resources [14]. Alternatively, the WCET bound of the original program can be derived without any code modifications by a computation which assumes the serial execution of all instructions, i.e., that there is no parallelism in the pipeline [18]. This approach is safe but can lead to gross overestimation of the WCET.

2.6 Step 3: WCET Bound Calculation

After program source-code and low-level platform analysis the next step in WCET calculation is the search for the program path that has the longest execution time. In the CFG representation of the program, where each edge has a corresponding weight i.e. execution time, the calculation step would search for the heaviest path.

The sources of overestimation in the calculation step are:

1. that not all infeasible paths have been found in the data-flow analysis of program source code

2. that some assumptions about the low-level timing effects had to be pessimistic because of safety

3. that possibly not all flow fact constraints can be modelled with the method used for the calculation step

The three principal methods used for the calculation step are: the *path-based*, the *tree-based*, and the *implicit-path-enumeration-technique* based method.

2.6.1 Path-based calculation

In *path-based* calculation methods the WCET is obtained by finding the longest path out of the set of all possible program execution paths. Not every program path is also a possible execution path. One technique that can be used for finding the longest feasible path in a small program is exhaustive path enumeration.

However, for large programs it is impractical to test all program paths for executability, since the number of possible paths rises exponentially with the number of control flow branches. Instead, the following simple heuristic can be employed according to [21]. The k longest program paths, where k is the number of control flow branches, are analyzed for executability. Using this method, the longest feasible executable path would be returned as the WCET inducing path. If all of the k paths are unfeasible, the WCET would be bounded by the shortest of the k paths.

The path-based WCET calculation is mainly employed on *straight-line* code which assumes the absence of loops and recursive function calls. For code without these assumptions, path-based calculation is impractical because of the exponential path growth. Automatic code synthesis tools for control applications often produce code that fullfills these assumptions, i.e. straight line code.

One of the widely used tools of this sort is SCADE. It enables *graphical programming* of control systems [22]; the SCADE compiler can automatically generate straight-line C code from the SCADE *graphical model* [23, p.23]. The WCET of such code is easily verifiable using the path-based WCET calculation technique such as the one presented in [21].

2.6.2 Tree-based calculation

In *tree based* calculation methods the final WCET is estimated by a hierarchical *bottom-up* traversal of the program syntax tree. The WCET of each tree is calculated from the WCET of all of its subtrees; on the lowest level, the WCET of each *tree leaf* is calculated directly from the WCET of its basic blocks.

2.6.2.1 Basic tree-based WCET calculation

Simple WCET calculation formulae for a processor without caches, pipelines, and branch prediction have been presented in the original paper [24]. They operate on four basic constructs of an imperative programming language: *primitives*, *sequences*, *if-alternatives*, and *for-loops*, from which more complex constructs such as *switch-case-alternatives*, *do-while-loops*, etc. can be composed.

Primitives Primitives S_i are the smallest constructs in the analyzed program; they cannot be subdivided into other constructs of a high level programming

language. Idealistically, the WCET of a primitive can be obtained by summing up the WCETs of the machine instructions that are contained in it; each primitive corresponds to a basic block in WCET analysis. The worst case execution times $WCET_S_i$ of all primitives S_i have to be known in order to perform the subsequent tree-based calculations.

$$WCET(S_i) = WCET_S_i \tag{2.3}$$

Sequences For *sequences* $S = [S_1, \ldots, S_n]$ the combined WCET is calculated as:

$$WCET(S) = \sum_{i=1}^{n} WCET(S_i) \tag{2.4}$$

Alternatives For *alternatives* of the kind $S = [\texttt{if}\ (test)\ \texttt{then}\ S_1\ \texttt{else}\ S_2]$ the combined WCET is calculated as:

$$WCET(S) = WCET(test) + \max(WCET(S_1), WCET(S_2)) \tag{2.5}$$

Loops For the bounded *for-loop* construct $S = [\texttt{for}\ (;\ test;\ incr)\ S_1]$ the WCET is calculated as:

$$\begin{aligned} WCET(S) = {} & loop_bound \times (WCET(test) + WCET(S_1) + WCET(incr)) \\ & + WCET(test) + WCET(exit) \end{aligned} \tag{2.6}$$

2.6.2.2 Extended tree-based WCET calculation

A more sophisticated approach for tree-based calculation with support for modern processors features has been presented in [2]. The central concepts of this work are *calling contexts* for functions and *hierarchical naming* for nested loops.

The WCET of each basic block depends on the execution path leading to that basic block. Thus, the same basic block in a function called from two different contexts can have two different worst-case execution times. Calling contexts distinguish between two different call sites of a single function by representing them as two different logical instances of the same function in the intermediate tree representation of the program. This is achieved by a compiler technique known as *function call inlining* that replaces the function call site with the body of the callee in the syntax tree representation. The same functions and, consequently, the same basic blocks can appear more than once in the logical representation of the program.

In nested loops the WCET of a basic block depends on the surrounding loop. Hierarchical loop names model this effect by giving each nested loop a name

prefixed with the name of the surrounding loop. Thus, each loop is assigned a certain *nesting level*. For example, if the nesting level of the outermost loop is named (0), then the levels of its n immediate nested loops would be named $(0,0),(0,1)\dots(0,n)$.

In order to support nesting levels the addition (+), and multiplication ($\times$) operators from the basic formulas are redefined with the $\oplus^L$, and $\otimes^L$ operators. Intuitively, the operator $\oplus^L$ sums up the WCETs of constructs S_i whose nesting level L_i is smaller or equal to L. The $\otimes^L$ operator is analogous; it multiplies and then sums up the WCETs of constructs S_i, whose nesting levels L_i are smaller or equal to L. By using these new operators, the WCETs of the surrounding program constructs can be calculated from the inner constructs (e.g. inner loops or calls to other functions) in a hierarchical and context-sensitive manner.

Furthermore, the extended tree-based model distinguishes $WCET^{seq}$ - a sequence of consecutive basic blocks - and $WCET^{jmp}$ - basic blocks whose outgoing edge is a jump to another more distant block. Basic blocks ending with the jump instruction can have a different execution time from consecutive basic blocks and can not be assumed to fall into the same locality category L.

The extended tree-based formulas operate on the same constructs as the basic formulas: *primitives*, *sequences*, *if-alternatives* and *for-loops*, but with the additional information about context level L and the basic block leaving edge (sequence or a jump).

Primitives For a basic block primitive S_i the WCET is independent from its context level L_i, since the basic block does not have any further inner elements. The extended formulas distinguish two cases depending on whether the primitive ends with the jump or not.

$$\begin{aligned} WCET(S_i, L)^{seq} &= WCET_S_i^{seq} \\ WCET(S_i, L)^{jmp} &= WCET_S_i^{jmp} \end{aligned} \tag{2.7}$$

Sequences For a *sequence* $S = [S_1, \dots, S_n]$ with the corresponding context levels $[L_1, \dots, L_n]$, the operator $\oplus^L$ sums the WCETs of all S_i whose level L_i is smaller or equal to L. If the sequence does not end with a jump, its WCET is calculated as:

$$\begin{aligned} WCET^{seq}(S, L) = WCET(S_1, L_1) \oplus^L \dots \\ \oplus^L WCET(S_{n-1}, L_{n-1}) + WCET^{seq}(S_n, L_n) \end{aligned} \tag{2.8}$$

For a *sequence* that ends with a jump, the WCET is calculated as:

$$WCET^{jmp}(S,L) = WCET(S_1,L_1) \oplus^L \dots \oplus^L WCET(S_{n-1},L_{n-1}) + WCET^{jmp}(S_n,L_n) \tag{2.9}$$

Alternatives For *alternatives* of the kind $S = [\texttt{if } (test) \texttt{ then } S_1 \texttt{ else } S_2]$ the combined WCET on context level L is calculated as:

$$WCET(S,L) = \max(WCET(S_1) \oplus^L WCET^{seq}(test,L), WCET(S_2) \oplus^L WCET^{jmp}(test,L)) \tag{2.10}$$

Loops For the bounded *for-loop* construct $S = [\texttt{for } (; test; incr)\ S_1]$ the WCET on context level L is calculated as:

$$WCET(S,L) = loop_bound \otimes^L (WCET(test,L) \oplus^L WCET(S_1,L) \oplus^L WCET(incr,L)) + WCET^{seq}(test,L) \oplus^L WCET(exit,L). \tag{2.11}$$

2.6.2.3 Pros and cons of tree-based calculation

The advantages of tree-based WCET calculation are its relative implementation simplicity, a hierarchical approach that operates on the program syntax tree, and the use of exact calculation formulae. The requirement that *unbounded recursion* is not allowed in the source code of analyzed programs is usual in static WCET analysis.

However, the hierarchical approach necessitates the restriction of analyzed source code to the *structured programming* paradigm. Statements that can cause unstructured breaks in the control flow such as `break` and `goto` statements in C are forbidden [2]. Thus, tree-based methods may not be able to analyze the output of automatic code generators, e.g. those for *Simulink*[1] or *TargetLink*[2], which sometimes generate unstructured code.

The challenges of applying the tree-based method to structured programs are how to properly express non-local flow information and how to achieve tight WCET estimates on processors with variable instruction timing [25]. The tree-based calculation method cannot express all types of program flow constraints, which can lead to pessimism in the WCET calculation [4].

[1] www.mathworks.com
[2] www.dspaceinc.com

2.6.3 IPET-based calculation

The *implicit-path-enumeration-technique* (IPET) is a WCET calculation technique that avoids the problem of exponential path blowup inherent in explicit path enumeration. IPET models the problem of finding the worst-case execution time of a computer program as a standard optimization problem of *integer-linear-programming* (ILP). Standard solvers for ILP, e.g. the open source *lp_solve*[3], are readily available. Theoretically, IPET has exponential execution time. However, this worst case does not occur in practice [26].

2.6.3.1 The standard ILP problem

The standard ILP problem consists of finding integer values for variables $x_1, \ldots, x_n$, so that the *objective function* F is maximized, while at the same time satisfying a set of *constraints* $C_1, \ldots, C_m$.

Variable x_i denotes the amount of some *resource* i. The *cost* of the resource is given as c_i. The m constraints C_j can be any linear combination of the variables x_i multiplied by constant factors a_{ij} together with the corresponding *binary relation* $\preceq$ and the *bound* b_j.

Optimization variables x_i

$$x_1, \ldots, x_n \tag{2.12}$$

Objective function $\boldsymbol{F}$

$$F = \sum_{i=1}^{n} c_i x_i \tag{2.13}$$

Constraints $C_1, \ldots, C_m$

$$C_j = \sum_{i=1}^{n} a_{ij} x_i \preceq b_j \, , j \in \{1, .., m\}, \tag{2.14}$$

where $\preceq$ can be one of the following binary relations: $\leq, <, \geq, >, =$

2.6.3.2 Encoding of the WCET problem

The optimization variables $x1, \ldots, x_n$ now denote the number of executions of basic blocks $1, \ldots, n$. Each x_i is bounded from above by a set of flow-of-control constraints. These constraints need to be obtained by program code analysis. *Structural constraints* are less difficult to extract; they can be derived directly from the program CFG using Kirchhoff's rule of flow conservation. The rule states that

[3] lp_solve - lpsolve.sourceforge.net

the sum of ingoing and outgoing flow has to be equal for each basic block, i.e. vertex of the CFG. *Semantic constraints* are more difficult to obtain; for them, special *data-flow* and *control-flow* analysis techniques are necessary.

In the following, two example programs are given on how to obtain the structural flow constraints. The first program consists of a single *if-then-else* alternative, and the second contains a single *while* loop. In both examples, the costs $c_1, \ldots, c_n$, i.e. the worst-case execution time of basic blocks, are assumed to be provided beforehand and to have a constant numeric value. The overall WCET of each program is then equal to the value of the maximized objective function F (Equation 2.13).

2.6.3.3 Example program: if-then-else alternative

The source code of the first analyzed program is shown in Listing 2.2. The corresponding CFG of the program is displayed in Figure 2.5.

Listing 2.2: A program containing a single *if-then-else* alternative. The associated basic blocks B_i are given in comments.

```
if (p) { //B1
  q := 1; //B2
}
else {
  q := 2; //B3
}
r := q; //B4
```

The structural constraints on the execution counts x_1, .., x_4 of program basic blocks can be obtained by following the Kirchhoff's rule of flow conservation. They are shown in Equations 2.15a to 2.15h.

$$e_1 = v_1 \tag{2.15a}$$
$$e_2 + e_3 = v_1 \tag{2.15b}$$
$$e_2 = v_2 \tag{2.15c}$$
$$e_4 = v_2 \tag{2.15d}$$
$$e_3 = v_3 \tag{2.15e}$$
$$e_5 = v_3 \tag{2.15f}$$
$$e_4 + e_5 = v_4 \tag{2.15g}$$
$$e_6 = v_4 \tag{2.15h}$$

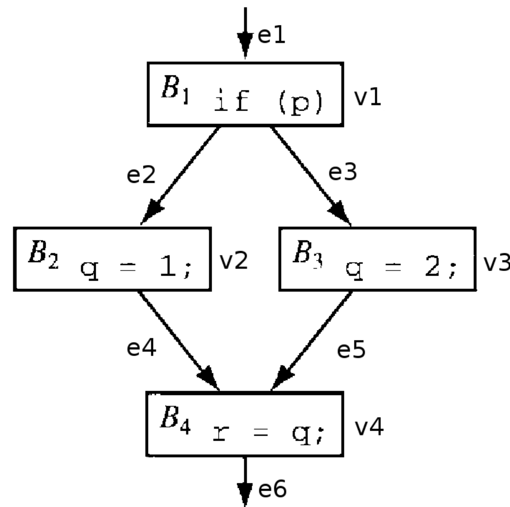

Figure 2.5: CFG of the *if-then-else* alternative [26].

In order to get numerical and not only symbolic constraints, the value of e_1 needs to be provided beforehand, either by the user, or automatically. For example, if the code fragment in Listing 2.2 were located at the beginning of a program main function, then e_1 should be equal to one. If however, the whole code fragment was located inside a loop, then e_1 should be equal to the maximum number of possible loop executions, i.e. the *upper loop bound* (Subsection 2.4.3.1).

2.6.3.4 Example program: while loop

The source code of the analyzed program is given in Listing 2.3. The corresponding CFG of the program is given in Figure 2.6.

Listing 2.3: Program with a single *while* loop. The associated basic blocks B_i are given in comments.

```
q := p; //B1
while (q < 10) { //B2
  q++; //B3
}
r := q; //B4
```

As in the previous example, the set of constraints (Equations 2.16a to 2.16i) is obtained by following Kirchhoff's rule of flow conservation between the vertices (basic-blocks) and edges of the CFG. In order to numerically solve the equations, the values of the entry edge e_1 and of the loop bound B need to be specified beforehand.

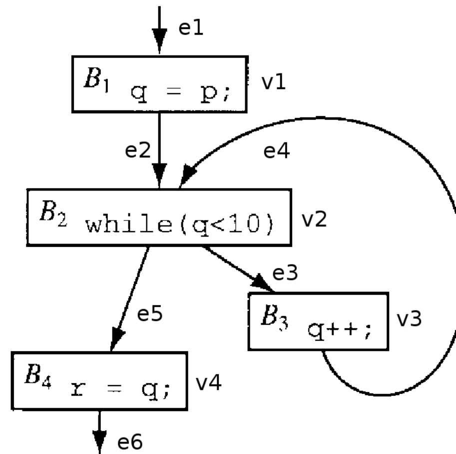

Figure 2.6: CFG of a *while* loop [26].

$$e_1 = v_1 \tag{2.16a}$$
$$e_2 = v_1 \tag{2.16b}$$
$$e_2 + e_4 = v_2 \tag{2.16c}$$
$$e_5 + e_3 = v_2 \tag{2.16d}$$
$$B \cdot e_2 \geq v_2 \tag{2.16e}$$
$$e_3 = v_3 \tag{2.16f}$$
$$e_4 = v_3 \tag{2.16g}$$
$$e_5 = v_4 \tag{2.16h}$$
$$e_6 = v_4 \tag{2.16i}$$

2.6.3.5 Pros and cons of IPET

The challenge with IPET is to obtain tight bounds for v_i and c_i, since otherwise pessimism in the WCET estimate is the result. If c_i is assumed constant for all basic block executions, then this can be a source of pessimism, e.q. when modelling a processor that possesses cache memory. The same basic block can have different execution times, depending on whether there is a cache miss or a cache hit. If c_i always assumes a cache miss, this results in undue pessimism of the WCET estimate.

Another source of pessimism in IPET arises from the encoding of *loop bound* constraints (Subsection 2.4.3.1). When a loop bound B is encoded as a constraint C_j, this results in a WCET estimate based on the assumption that the loop will

always execute for the maximum number of iterations as specified by the bound. However, the maximum number of loop iterations may vary according to the context in which the loop is placed. Depending on its context, each loop can have several upper bounds. Using a single upper bound that is valid for all contexts necessarily makes the bound pessimistic and induces overestimation.

In spite of these difficulties, IPET is reported to be one of the most flexible technique for modelling flow constraints in WCET calculation [25].

2.7 Types of WCET Analysis

A possible division of WCET analysis methods is one based on how timing information is derived. If it is derived 'offline' from a *processor model*, the method in question belongs to the class of static WCET analysis methods. If timing information is instead obtained 'online' by execution-time *measurements* on a real computer system, then measurement-based WCET analysis is implied.

Static and measurement-based methods both have the following steps in common: *static source-code analysis* and WCET *bound calculation*. During source-code analysis, both the static and measurement-based methods perform *control-flow analysis*. Static methods additionally perform the more complex *data-flow analysis* as well. For the WCET bound calculation both methods can, in principle, use any of the techniques discussed in Section 2.6.

One example of differences between the two methods is processor modelling, which is exclusively relevant for static WCET analysis. In contrast, standardized and architecture independent *measurement interfaces* are only relevant for measurement-based analysis.

Depending on the basic unit of source code under consideration, measurement based methods can be further subdivided into *end-to-end measurement* and *hybrid* methods. In end-to-end measurement the basic unit of source code is the whole program, while hybrid methods deal with smaller units, e.g. *program segments*.

Common to all measurement based WCET methods is that they need to generate *test data* whereby to execute the software under test in order to perform measurements. The generation of test data can be performed randomly or by using heuristics. The main advantage of heuristics is that they perform the search in a smart way, whereas random search operates purely by chance. Some heuristics can systematically search for the longest execution path by a feedback-based improvement of the test data [25].

Hitherto genetic algorithms have often been used for measurement based WCET estimation, for example in [27]. They have also been used for automated functional testing of generic computer programs [28], as well as for testing real-time systems [29] and safety-critical systems [30]. The method presented in Chapter 4 of this

work generates test-data by a novel *particle swarm optimization* heuristic.

2.7.1 Static WCET analysis

The steps of static WCET analysis are the following:

1. Static code analysis and partitioning of the program into segments
2. Derivation of segment execution times
3. Calculation of the WCET bound from the combined segment execution times

2.7.1.1 Static code analysis

Static code analysis is a well developed technique from traditional compiler construction. One of its concepts that is relevant for WCET analysis is the CFG representation of a program. The CFG explicitly states the set of all statically possible execution paths in a program. However, not all of these paths are dynamically possible at run-time and, therefore, all of them cannot contribute to the overall program execution time. As analyzing all execution paths is computationally expensive, infeasible paths should be eliminated from WCET analysis. There are two approaches for reducing the number of execution paths under consideration.

The first approach is to analyze the program data-flow and dependencies between the variables in order to extract *flow-facts*, i.e., determine which execution paths are not feasible. The techniques used for such analysis are implemented in modern, code optimizing compilers and in tools for static verification. Detailed information about static code analysis can be found in books on compiler engineering and program analysis, e.g. in [31, 32].

The second approach for reducing the number of potential execution paths is segmentation. The program is partitioned into *segments* where each segment contains a certain number of basic blocks. Only paths passing through the segments, not through the basic-blocks, need to be subjected to further feasibility checks. This reduces the overall number of paths that need checking.

Many techniques which are used to determine whether a certain path is feasible or not come from the theories of *static verification* and *formal methods*. One of these techniques is *abstract execution*, which executes the program using a reduced number of states.

For example, a program consisting of a single loop with a 32 bit integer counter can have 2^{32} statically possible states. If the exact value of the *loop counter* is not of interest, abstraction can reduce the 2^{32} possible states to only two states, namely *loop counter* $<$ *loop limit* or *loop counter* $\geq$ *loop limit*. The challenge

is to find the minimum value for *loop limit* so that the value of the dynamic *loop counter* is always below it (Subsection 2.4.3.1). Other more useful abstractions, e.g. Presburger arithmetics, are available from the theories of formal methods.

2.7.1.2 Derivation of timing information

Unlike static source code analysis, which uses many techniques developed in other areas of informatics, the derivation of timing information for *instructions*, *basic blocks*, *segments*, and *execution paths* is relevant almost exclusively to WCET analysis.

The central issue of static WCET analysis is how to construct a *processor model* that can be used to yield safe and reasonably tight worst-case execution time estimates of the categories numbered above. Up to date there exists no *systematic* way of constructing such a model [25]. The challenges are twofold: first is the inherent complexity of modern processors where the execution time of each instruction can be influenced by the instructions preceding it and by the specific input-data that the instruction receives. Second, it cannot be guaranteed that the same initial state of the processor will always produce the worst-case instruction execution time (Subsection 2.5.4). In addition to that, the *closed source* nature of many commercial processors presents a further challenge in modelling their temporal behavior.

2.7.1.3 Calculation step

The techniques used for the calculation step are not unique to static WCET analysis; they can also be applied for hybrid WCET analysis. Static WCET analysis may use any of the three *calculation techniques* presented in Section 2.6. *Explicit-path-enumeration* is usually avoided, since it is too computationally expensive. Instead, *tree-based-calculation* and IPET are preferred.

2.7.2 End-to-end measurement

End-to-end measurement is reported to be the currently most widely used method for WCET estimation in industry [25]. This is mostly due to its simplicity, i.e. absence of any program or *execution platform* modelling. End-to-end measurement does not provide safe WCET bounds, since there is always a possibility that not all execution paths are exercised by the generated test data. Since the number of potential execution paths rises exponentially with the size of the program, end-to-end measurement quickly 'runs out of steam' when applied to larger programs [4].

2.7.3 Hybrid WCET analysis

The basic feature of methods for *hybrid WCET analysis* is that source code analysis and WCET bound calculation are performed by techniques similar to those used in *static WCET analysis.* Timing information, on the other hand, is derived by direct measurements on a processor system. Hence, methods for hybrid WCET analysis share characteristics of both static WCET analysis and *end-to-end measurement.*

2.7.3.1 Program segmentation

The chief task of *static code analysis* for hybrid WCET methods is the partitioning of the program into segments. There are two extremes cases.

> *Case 1*: Each program segment consists of one basic block. This results in the maximum number of program segments; each segment has only one execution path inside.
>
> *Case 2*: The whole program is one big segment. Since all basic blocks are in one segment, the number of paths tends to be intractable for large programs.

Only paths within each segment are relevant for the segment's worst-case execution time. For a segment of reasonable size, it is relatively easy to explicitly enumerate all paths inside it. The total number of execution paths is a function of the number of segments. The number of segments itself is determined by the maximum number of paths per segment i.e. the *path bound* [33]. Figure 2.7 illustrates this relationship between the number of segments and the total number of paths.

It is the goal of program partitioning to balance the number of segments with the average number of paths per segment. The necessary instrumentation of each segment introduces computational overhead. Up to the point of global minimum, the number of paths that need to be evaluated is inversely proportional to the number of segments. Analyzing the feasibility of paths within segments and measuring their execution times can be computationally intensive and time consuming [33].

2.7.3.2 Feasibility of execution paths inside segments

After partitioning a program into segments, the next subtask of program code analysis is to determine the feasible out of the set of possible paths for each segment.

Each segment spawns a certain number of execution paths that pass through it. Program execution can be selectively guided to those paths in order to analyze the

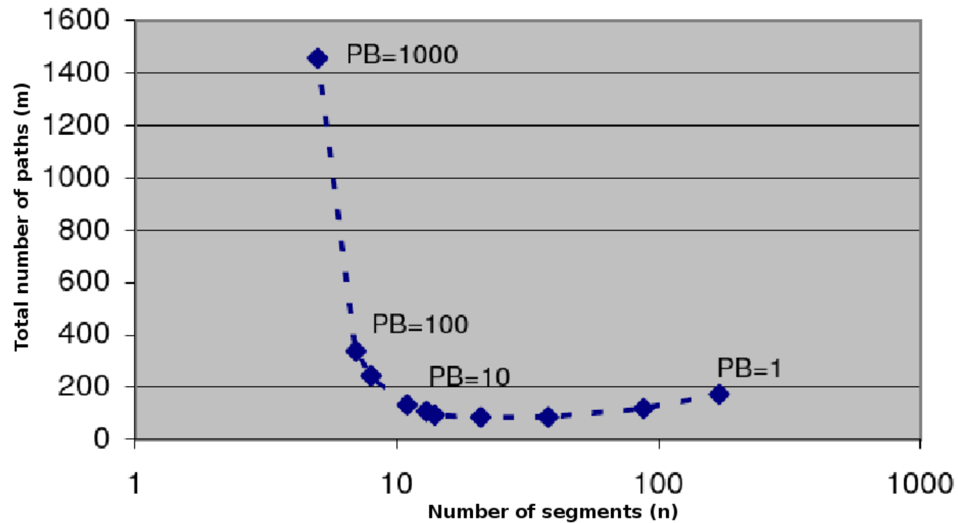

Figure 2.7: Relationship between the number of program segments n and the total number of execution paths inside segments m. For one segment per program, the total number of paths is maximal. Then, with the growing number of segments, the total number of paths quickly falls, until for a certain number of segments it achieves the minimum. After that, the number of total paths slowly rises until a local maximum is achieved. Each segment consists of one basic block for a path bound (PB) of one. [33]

worst-case execution time of the segment. The guidance can be achieved by using a heuristic optimization algorithm to generate input data. The fitness function of the algorithm should give higher preference to input data that exercise paths spawned by the analyzed segment than to other paths.

The paths for which no input data could be found can afterwards be subjected to model checking, which ultimately determines for each path whether it is feasible of not. Since model checking is computationally expensive, it is only possible to use this technique on a limited set of paths after the majority of the feasible paths were already found by heuristics [33].

2.7.3.3 Derivation of timing information

After program segmentation and feasibility checks of the execution paths it is possible to derive the *timing information* of segments. The hybrid WCET method derives timing information by executing the program with varying test data and measuring the execution times of the subpaths spawned by each segment. At the

end of this process each segment is associated with the longest *observed* execution time of any of its subpaths.

2.7.3.4 Calculation of the WCET bound

In the *calculation step*, the worst-case execution paths of program segments are combined into the *WCET bound* of the whole program. In hybrid WCET analysis the bound calculation is often performed with the IPET method, e.g. in [33].

2.7.3.5 Conclusion

Hybrid WCET analysis reduces the complexity of *path coverage* inherent to end-to-end measurement by partitioning a program into segments. At the same time it retains the advantages: it completely avoids the resource-intensive and time-consuming construction of a processor model by deriving timing information from target hardware. Heuristic algorithms are applied only for finding the feasible paths inside segments, and not, as it is the case in end-to-end measurement, for the more difficult problem of finding the WCET inducing execution path [33].

2.8 Overview of WCET Tools and Related Work

Commercial and academic WCET tools range from completely or mostly static to completely measurement based tools. Examples of completely static tools are: aiT, Bound-T, Chronos, Heptane, and the static prototype of TU Vienna. Examples of mostly static tools are: Sweet, SymTA/P, the prototype of TU Chalmers, and the hybrid prototype of TU Vienna. Completely measurement based tools are: RapiTime and the measurement-based prototype of TU Vienna.

2.8.1 Completely static WCET tools

2.8.1.1 aiT

aiT is a commercial WCET tool from AbsInt Angewandte Informatik in Saarbrücken, Germany. It is a static tool that obtains timing information from a processor model. Steps of aiT are: (1) *static flow analysis*, (2) *value analysis*, *cache/pipeline analysis* and (3) WCET *bound calculation*.

Static flow analysis is done on the object-code level. The result is a CFG annotated with automatically derived *flow facts*. These flow facts include information about *structurally infeasible paths*.

After static flow analysis, *abstract interpretation* is used for cache/pipeline and value analysis. As far as it is statically feasible, value analysis determines value

ranges of possible register contents at every program point. This information is used for obtaining *static loop bounds* and *semantically infeasible paths*.

Flow facts about input *dependent loop bounds* and infeasible execution paths that were missed in the previous analysis have to be provided by the user. The user can specify annotations at the source code level or in a separate configuration file.

Cache and pipeline analysis is based on a processor model and is carried out using abstract execution Cache analysis supports caches with LRU, *round robin*, and pseudo-LRU replacement policies. Non-LRU policies have reduced predictability when compared with perfect LRU policies. Pipeline analysis gives the WCET of each basic block in the context of its execution inside the pipeline.

In the last step aiT calculates the total WCET bound. The step is designated as path analysis, since path constraints are formulated as an ILP problem with IPET.

The aiT tool supports a multitude of processors; among these are the PowerPC MPC, Motorola ColdFire, ARM7, Renesas M32C-85, and Infineon TriCore processors. [19, 36:22]

2.8.1.2 Bound-T

Bound-T is a tool for static WCET analysis developed at Tidorum Ltd., Finland.

The tool takes a binary executable and a user annotation file as input; it produces the WCETs of program subroutines (with contained function calls) and the call and control-flow graphs as output. Bound-T is used by ESA[4] in its tool chain for the verification of hard real time on-board software.

The processor model of Bound-T is constructed using information from the processor manuals. The tool models the flow-of-control and integer-arithmetic, but it lacks general support for more complex features such as concurrent operation of integer and floating-point units and register overflow/underflow traps.

Basically, the tool has the following work-flow. First, limited flow analysis detects some unfeasible program paths and loop bounds. Information about the unfeasible paths and loop bounds that were missed has to be provided by the user. After flow analysis, the WCET problem is modelled by IPET and solved by a standard ILP solver.

An innovative concept of Bound-T is its use of the CFG as a *pipeline state graph*. Here, nodes represent different pipeline states, and edges represent timing dependencies between instructions. This CFG clearly models the *before-after* relation of the instructions' effects on variables (registers). Thereby, the tool can

[4]ESA - European Space Agency

express complex interdependencies between the computational effects of instructions.

The limitations of Bound-T are twofold. First, input programs have to follow an enforced programming style in order to conform to the patterns implemented in the *control-flow analysis*. Secondly, the tool currently supports only *cacheless* or processors with very small caches. Altough such processors have the disadvantage of lesser performance, their advantage is simpler modeling and the absence of timing anomalies.

Bound-T currently supports the Intel-8051 series, Analog ADSP-21020, Atmel ERC32, Renesas H8/300, and the prototypes of ARM7, ATMEL AVR and ATmega processors [19, p.36:24].

2.8.1.3 Chronos

Chronos is a static, open-source tool for WCET analysis developed at the National University of Singapore.

It supports complex processors with *out-of-order pipelines*, *instruction caches* and *dynamic branch prediction*. Furthermore, the tool accounts for the timing effects due to interaction of the above features.

Chronos uses the SimpleScalar toolset, which is a cycle accurate *processor simulator*. The toolset can simulate many processor features: different cache replacement policies, various branch prediction schemes, etc. Chronos can analyze the WCET on a multitude of processors as long as they can be simulated in SimpleScalar.

The input to Chronos is a program task written in C and the processor configuration for SimpleScalar. The CFG is constructed on the object-code level, while *user annotations* have to be specified on the source code level. Annotations are needed for bounding the maximum number of loop iterations and for information about infeasible execution paths. Chronos can detect some *loop bounds* by simple *data-flow analysis*; undetected loop bounds have to be specified by the user.

The mode of operation of Chronos is the following: using the simulated processor model, the WCET of each basic block in the CFG is bounded for each possible execution context. *Execution contexts* can be branch misspredictions in preceding basic blocks or instruction cache misses in the current basic block. Afterwards, the highest number of basic block executions in each corresponding context is bounded by ILP. Provided with this information and linear flow constraints (loop bounds, unfeasible paths), the final WCET of the program is calculated with IPET.

Currently, Chronos does not support data-caches; it always assumes a miss on data-cache access. In some cases, this can lead to pessimistic WCET estimates. [19, 36:29]

2.8.1.4 Heptane

Heptane is a static, open-source WCET analyzer that is developed at IRISA, Rennes. It can analyze C programs either on source or object code level.

The bound calculation on the source-code level, uses the tree-based structural approach as described in Subsection 2.6.2. The bound calculation on object-code level uses IPET (Subsection 2.6.3) and operates on the CFG extracted from the object-code.

Both calculation methods provide safe WCET bounds. The structural method is faster than the IPET method, but it delivers a more conservative WCET estimate. It has no support for compiler optimizations at the object-code level. These optimizations can create a mismatch between the CFG on the object-code level and the program syntax tree on the source-code level.

In Heptane, the information about loop bounds, mutually-exclusive and unfeasible path is provided exclusively by the user using *symbolic annotations* in source code. Currently, Heptane lacks any capability for static flow-analysis. The tool mainly concentrates on modelling hardware timing effects.

Heptane accounts for the effects of instruction-caches, pipelines and branch-prediction on program timing. Data-caches are currently not supported. All processor features are modelled using a *microarchitecture-independent formalism*, which enables Heptane to be easily retargeted to new architectures.

Currently, Heptane is limited to scalar monoprocessor architectures with in-order pipelines. Examples of supported processors are: Pentium 1, StrongARM 1110, Hitachi H8/300 and the virtual MIPS processors. [19, p.36:31], [2]

2.8.1.5 The static prototype of TU Vienna

The prototype of TU Vienna for static WCET analysis operates on *wcetC* - a subset of C - which contains language constructs for user annotations of infeasible execution paths. The prototype is integrated with a modified *GNU C* compiler to keep track of the code optimizations performed during compilation.

An interesting feature of the prototype is the support for automatic timing analysis in the Matlab/Simulink toolchain. When operating as part of the toolchain, the prototype needs no user annotations, and it provides reasonably tight WCET bounds. When working on generic wcetC programs, the quality of the results depends heavily on the quality of user annotations.

The tool supports the M68000, M68360, and the C167 processors. [19, 36:26], [9]

2.8.2 Mostly static and hybrid WCET tools

2.8.2.1 Sweet

Sweet is a modular WCET analysis tool with an integrated research compiler. It is developed at the Mälardalen University of Technology in Sweden.

The tool's main components are flow analysis, the memory and pipeline analysis, and bound calculation. The integrated compiler enables the code analysis on the intermediate code level, thus avoiding differences between the analyzed and executed code.

Flow analysis is carried out by a combination of:

- *Program slicing* which restricts the analysis to only those program parts that can affect the flow-of-control;

- *Pattern matching* which can automatically detect loop bounds of simple loops;

- *Abstract execution* for the rest of the code.

Execution time analysis together with memory and pipeline analysis is carried out by a simulation on a cycle-accurate processor model. Currently supported are the ARM9 and NEC V850E processors. From these only the model of NEC V850E has up to now been validated against real hardware.

The bound calculation is implemented by using three different calculation techniques: the quick path-based, the global IPET-based and a *hybrid clustered technique*, which is combination of local IPET and local path-based calculation.

Sweet currently supports the analysis of in-order pipelines and one-level instruction caches. [19, p.36:34], [13, p.443]

2.8.2.2 Prototype of the Chalmers University of Technology

The prototype of the Chalmers University of Technology in Sweden supports WCET analysis of the high performace *PowerPC* architecture with pipelines and multiple cache banks.

Two notable features of the prototype are the use of a cycle accurate processor simulator for symbolic code execution without any input data and binary code transformations which eliminate the potential for timing anomalies from dynamic instruction scheduling (Subsection 2.5.4).

The tool currently supports only a subset of the PowerPC instruction set; therefore the analysis is safe only for programs using those instructions. Furthermore, the tool is limited by the high computational requirements of the employed analysis techniques.

2.8.2.3 SymTA/P

SymTA/P is a tool for mostly static WCET analysis, developed at the Braunschweig University of Technology in Germany. It produces both the upper and lower bounds on the execution time of C programs.

SymTA/P tries to avoid the pitfall found in most static tools, that they are platform specific, by separating the flow analysis on source code level from the measurements on object-code level.

Flow analysis is platform independent; its result is the program CFG where each node corresponds to a *single feasible path* (SFP) sequence. An SFP sequence is a sequence of basic blocks which is invariant of program input data. SFP sequences reduce the number of instrumentation points needed for measurements, since now, instead of instrumenting each basic block separately, it is only necessary to instrument the beginning and the end of the SFP sequence.

The execution time measurement of each node in the CFG is obtained by either simulating the program on a cycle accurate processor simulator or by executing it on an evaluation board. The tool supports static analysis of data-cache accesses and symbolic execution is used to identify the input-dependent memory accesses. The timing behavior of such accesses is bounded with ILP.

The tool supports the execution time analysis of multiple processor architectures, e.g. ARM9, TriCore, StrongARM, C167, and i8051. [19, p.36:38]

2.8.2.4 The hybrid prototype of TU Vienna

The hybrid prototype developed at the TU Vienna *real-time-systems group*[5] is a combination of a static approach for source code analysis and bound calculation and a measurement based approach for the derivation of execution time.

During static code analysis, the program is partitioned into *segments* of reasonable size, each segment containing multiple basic blocks (Subsection 2.7.3.1). Furthermore, *feasible paths* within each segment are identified by a stepwise three-phase approach.

1. Input data generated by random search are used to exercise program paths within all program segments in order to easily identify a subset of all feasible paths.

2. A *genetic algorithm* is employed to selectively generate input data that exercise specific segments in order to find additional feasible paths.

3. *Model checking* is used to find feasible paths in each segment that were missed in the preceding two phases.

[5]http://ti.tuwien.ac.at/rts

The multiphase approach compensates for the computational expense of model checking alone; model checking needs to be applied only to a small number of potentially feasible paths. The execution times of program segments are obtained by executing the program with the generated input data on real hardware. After local measurements, the WCET of each segment is greedily combined into the global WCET by using IPET.

The downside of this method is that the compiler may not do any optimizations on the object-code level, since the method does its static code analysis on the source-code level. Furthermore, as with all measurement based methods this method is not safe for processors with timing anomalies. For such processors, greedily taking the WCET of parts of the program - in this case of program segments - does not guarantee to produce the WCET of the whole program.

The advantage of the hybrid method is the relative ease of porting to new architectures, which only requires modifying the measurement methodology. The method was applied to the HCS12 and Pentium processors. [19, p.36:29], [33]

2.8.3 Completely measurement based WCET tools

2.8.3.1 RapiTime

RapiTime is the commercial version of pWCET, a research tool developed at the University of York. The tool is completely measurement based, i.e., all basic block timing information is derived from measurements. Since the tool does not use a processor model for obtaining timing information, in principle, it can provide WCET bounds for any processor architecture. This only requires porting the object-code reader and the tracing mechanism whereby measurements are obtained.

RapiTime takes a program written in C, Ada, or an executable as input. Furthermore, users need to provide test data for measurements and annotations in program code. Annotations can be used to guide the WCET analysis of the tool and to determine execution contexts. Different execution contexts help to analyze inlined loops and different calls to the same function. RapiTime displays a html report as output; the report contains WCET predictions and measured execution times for each program function.

The WCET bounds provided by measurement-based tools are generally not safe, because of timing anomalies present in modern processors as discussed in Subsection 2.5.4. A novel concept of RapiTime which deals with this problem is the use of *probability distributions* of basic block measurements. Using a *probability algebra* on a program tree representation, basic block probabilities are combined into the WCETs of program functions and subfunctions.

It is believed by the authors of RapiTime that absolute WCET bounds of a

program are overly pessimistic if the probability of the worst case event happening is small. For example, if the probability of a missed deadline caused by a wrong WCET task prediction is of the same order of magnitude as other dependability estimates of the hard real-time system, e.q. (10^{-12}), then this is deemed as acceptable.

Processors currently supported by RapiTime are the Motorola MPC555, HCS12, ARM, MIPS, and NecV850. [19, p.36:38], [34]

2.8.3.2 The measurement-based prototype of TU Vienna

The prototype of TU Vienna for *measurement based* WCET analysis is an *end-to-end* measurement based (Subsection 2.7.2) approach that greedily maximizes the execution-time objective function.

The method works as follows: at the beginning, a population of input data vectors is initialized randomly and used as input for the analyzed program. The program execution time for each of the input is measured and stored. Based on this information each data vector gets a certain fitness value, depending on the length of execution. Data vectors which induce longer execution times are assigned higher fitness. Subsequent populations of input data are generated from the previous populations using a *genetic algorithm* (GA). The generation of new population and corresponding measurements are repeated until the termination criterion is reached.

This method does not guarantee a safe WCET bound; it bounds the WCET from the lower side of possible execution times. As with all measurement and hybrid methods, the porting to new processor architectures is relatively easy. The method was tested on the C167 and PowerPC processors. [19, p.36:27], [27]

2.8.4 Related work

The method presented in this bachelor thesis (Chapter 4)has the most similarity to the *end-to-end* measurement approach (Atanassov [27], Subsection 2.8.3.2) and to the *hybrid* approach (Wenzel [33], Subsection 2.8.2.4). The main differences are in the population generation strategy, timing information derivation, method of WCET calculation and in program partitioning. A detailed comparison is given in Section 4.7 of Chapter 4.

Chapter 3

Particle Swarm Optimization

3.1 Introduction

Particle swarm optimization (PSO) is a heuristic optimization technique that simulates information sharing between social animals, such as insects, flocks of birds, and others.

The basic idea of the PSO algorithm is that sharing certain information between the entities, like the location of food, creates special group dynamics, i.e. group intelligence, which carries a distinct evolutionary advantage for the whole group.

One animal by itself is not able to search a big area of space. However, by sharing its previous experiences about food locations with other animals, each animal increases its own information about the surroundings. Consequently, the probability of each animal finding food increases.

In the following discussion, the natural animals will be replaced by the concept of a massless, volumeless particle. Each particle occupies a point in *m-dimensional* space, has a certain velocity, and moves in a certain direction at discrete time steps.

The social simulation is achieved by the following simple rules:

1. The fitness of each particle is calculated by a *fitness function* that takes the current position of a particle as input.

2. The change of the *particle position* is calculated at every iteration by adding the updated velocity vector to the current position. This brings about the movement of particles in the search space.

3. The change of the *particle velocity vector* is determined by the previous velocity vector, the previous best position, and the group's best position. Particles that are immediate neighbors to each other form a *group*. The memory of each particle about its previous best position is called the history best (*hbest*) position; it is used to model simple nostalgia, a tendency of

particles to return to the place that most satisfied them in the past. The group or neighborhood best position (*gbest*) is a position with the highest fitness that was discovered by the particles of some group. The definition of a group depends on the definition of a topology, i.e., which particles are direct neighbors.

3.2 The Continuous PSO Algorithm

The continuous PSO algorithm (CPSO) uses the concepts from the above introduction in a straightforward way. It is an abstract simulation of a natural phenomenon: particles flying in the search space, combined with the rules of information sharing between living organisms.

3.2.1 The continuous search space

The continuous PSO algorithm is characterized foremost by operating inside an m-dimensional search space $\mathbb{X}^m$, where m is the number of dimensions, and $\mathbb{X}$ is a set containing numbers from some interval $I = [X_{min}, X_{max}]$. The constants X_{min} and X_{max} can be doubles in a C implementation. They store the minimum and maximum possible value of each particle coordinate. Since particle coordinates are encoded in computers with a finite number of bits, the interval I is by necessity countable. The continuous PSO algorithm can also operate on integer coordinates by rounding each double coordinate to its nearest integer value.

In this work a version of both the continuous and the discrete binary PSO algorithm is implemented. Since the discrete binary PSO algorithm is an extension of the continuous PSO algorithm, it is first necessary to discuss the concepts of continuous PSO. The discrete binary PSO is discussed in Section 3.3.

3.2.2 Definition of a PSO particle

The particle is the central object whereupon the PSO algorithm operates; however, its modelling can take various forms. From previous work on *particle swarm optimization*, e.g. [35, 36, 37], one could conclude that the concepts related to PSO particles: $fitness$, *position*, *velocity*, etc. should each be modeled as a separate array containing only one type of information for the entire particle swarm. Upon trying this approach, the information about a single PSO particle tended to become dispersed in many data structures.

To avoid this, a new data type for the PSO particle has been introduced in Listing 3.1. It defines a data structure that encapsulates all relevant information for a single PSO particle. Using it, the particle swarm is represented as an array

of PSO particles and not as a set of arrays where each array models a single PSO concept.

This approach reduces code coupling and gives a standard interface to the PSO particles. It makes it possible to write functions that operate directly on the *particle data type*, thus treating PSO particles as objects. In the author's experience during the practical implementation, this has resulted in more code reuse, easier configuration, and more freedom in experimenting with PSO extensions (Section 3.5). The pointers in Listing 3.1, e.g. $*$*position*, are implementation dependent; the reader may easily imagine them as object references.

Listing 3.1: Definition of a PSO particle

```
/* PSO particle */
typedef struct particle_s {
        position_t *position;
        position_t *velocity;
        double fitness;
        particle_linked_list_t *neighbor_particles;
        struct particle_s *hbest, *gbest;
} particle_t;

/* position in search space */
typedef union position_u {
        int pos_int[];
        double pos_double[];
} position_t;

/* particle swarm */
typedef struct particle_linked_list_s {
        particle_t *this;
        struct particle_linked_list_s *next;
} particle_linked_list_t;
```

3.2.3 Algorithm pseudocode

Algorithm 1 shows pseudocode of the continuous PSO algorithm. Each of the algorithm tasks are discussed in the following subsections.

3.2.4 Particle initialization

The important subtasks of particle initialization are the initialization of memory and the setting of initial particle values. The setting of particle values primarily

Algorithm 1 Pseudocode of the continuous PSO algorithm

```
 1: Procedure PSO_CONTINUOUS ()
 2: Initialize particles
 3: while terminate() is false do
 4:    For each particle: Evaluate position
 5:    Update neighborhood relation between particles
 6:    for each particle do
 7:       Determine if current position better than history best position
 8:       Determine group best position in particle's neighbourhood
 9:       Update velocity vector
10:       Update positions according to the velocity vector
11:    end for
12: end while
```

deals with the initial state of the particle's position, velocity, fitness, and the neighborhood relation.

3.2.4.1 Initialization of particle memory

Memory initialization includes the static reservation of memory space at the beginning, as well as providing functions for dynamic memory allocation during algorithm execution. Furthermore, functions for freeing memory should be implemented as a complement to initializing memory. After the reserved memory is no longer needed, it should be deallocated in order to avoid program memory leaks and operating system slowdowns or possible crashes. Exact details of memory allocation are relevant when the PSO algorithm is implemented in a low-level programming language without support for automatic memory management, e.g. in C, but they are not necessary for the understanding of PSO.

3.2.4.2 Initial particle position

When initializing particles it is desirable that their positions are equally distributed over the whole range in search space $\mathbb{X}^m$, so they can start searching for the optimum from different positions. The goal of position initialization is to achieve the greatest possible initial coverage of the search space. Usually, this is achieved by initializing the particle coordinates with random values from the interval $I = [X_{min}, X_{max}]$. The interval I thus defines the borders of the search space.

3.2.4.3 Initial particle velocity

The initialization of particle velocity vectors can basically be done in two ways. One can choose a steady state model and initialize all particles to zero velocity at the beginning. An alternative would be to randomly initialize the velocity vectors using a certain maximum velocity limit.

3.2.4.4 Initial particle fitness

Particle fitness can be set to any value, usually the minimum possible fitness. The initial fitness value does not influence the starting state of the computation, since it is immediately overwritten by position evaluation in the next step.

3.2.4.5 Initial particle neighborhood relation

The *neighborhood relation* between particles is needed for determining the group best position (*gbest*). Depending on the used topology (discussed in Subsection 3.4.1), each particle can have a different number of neighbors. This starts from two neighbors per particle in the *circle topology* up to all particles being neighbors in the *star topology*. The neighborhood relation can change at runtime which motivates modelling the neighbors of each particle as a dynamic linked list in Listing 3.1.

3.2.5 Particle evaluation

The evaluation step in the continuous PSO algorithm deals with the calculation of fitness, based on the current position of the particle. The calculation is performed by a fitness function which is actually the objective function of the problem that needs to be optimized.

$$particle_fitness := fitness_function\,(particle_position) \tag{3.1}$$

3.2.6 Update of particle history best position

Each particle has a certain position called history best or simply *hbest* with the highest fitness among all previous positions of a particle. This position represents the particle's memory of travelling through the search space. It is used in subsequent steps of the algorithm to influence the update of the particle's velocity vector. For convenience the *hbest* position can be encapsulated in a separate *particle type*. It is updated in every iteration by the following rule:

$\textbf{if}\;\; (particle \rightarrow fitness) > (particle \rightarrow hbest_particle \rightarrow fitness)\;\; \textbf{then}$

$particle \rightarrow hbest_particle := particle$
end if

3.2.7 Update of particle group best position

In contrast to the *hbest* position, that is unique to each particle, the group best position (*gbest*) is the current best position among a group of particles. Which particle belongs to which group is defined by the *neighborhood relation*. The *gbest* position influences the update of the velocity vector of each particle in a group. It is determined by the following rule:

for $g \in (particle \rightarrow neighbor_particles)$ **do**
if $(g \rightarrow fitness) > (particle \rightarrow fitness)$ **then**
$particle \rightarrow gbest_particle := g$
end if
end for

3.2.8 Update of particle velocity

After updating the *hbest* and *gbest* position, the velocity vector of each particle is also updated. The new velocity vector is a function of the old velocity vector, the old particle position, and the recently updated particle's *hbest* and *gbest* position. The rule for the update of the velocity vector is the following:

$$\vec{v}^{\,t+1} = \vec{v}^{\,t} + \overbrace{c_1 \cdot rand_1 \cdot \underbrace{(\vec{hbest} - \vec{p}^{\,t})}_{v_{\vec{hbest}}} + c_2 \cdot rand_2 \cdot \underbrace{(\vec{gbest} - \vec{p}^{\,t})}_{v_{\vec{gbest}}}}^{\Delta\vec{v}^{\,t+1}}, \tag{3.2}$$

where $c_1 = cognitive\ constant$,
$c_2 = social\ constant$

The factors c_1 and c_2 in Equation 3.2 are the user defined *cognitive* and *social constant*. Sometimes they are also called the *learning factors*. The random factors $rand_1$ and $rand_2$ can be any rational number in the interval $[0.0, 1.0]$. The value of 2 is often taken for the cognitive and social constant, since this gives the mean of 1 when it is multiplied with the random factors. Using these settings, the particles tend to 'overfly their targets half of the time' [35] and, thus, explore the space beyond and between the known local optima. This ability is responsible for at least some success of the PSO algorithm [38].

The random factors used in the calculation of the new particle velocity vector introduce indeterminism into the PSO simulation. What is known about the vector

$\Delta\vec{v}^{t+1}$ is that it will be oriented somewhere in the direction between $\vec{hbest}$ and $\vec{gbest}$. Its exact orientation and value depend on the random terms, which is sketched in Figure 3.1.

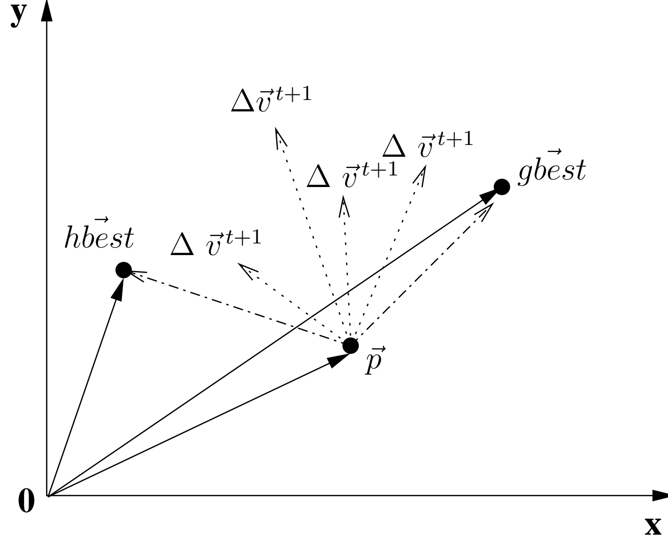

Figure 3.1: Change of the velocity vector $\vec{v}^{t+1}$ for a hypothetical particle p with position $\vec{p} \in \mathbb{X}^2$. More than one possible $\Delta\vec{v}^{t+1}$ illustrates the effect of random terms $rand_1$ and $rand_2$ in Equation 3.2.

3.2.8.1 Limiting particle velocity

Generally, the expression in Equation 3.2 allows for unbounded velocities. Thus, after a sufficient number of iterations, particle velocity could reach values approaching the diameter of the whole search space. This would make The granularity of such movement would be too great to effectively explore the defined search space. Thus, it is necessary to limit the particle velocity as shown in Equation 3.3. Each component v_i of the particle velocity vector $\vec{v}$ is limited with a scalar V_{max}.

$$limit_v(v_i) = \begin{cases} v_i & \text{if } |v_i| \leq V_{max} \\ \text{sign}(v_i) \cdot V_{max} & \text{otherwise} \end{cases} \tag{3.3}$$

It is reported in [39] that good results are obtained with a setting $V_{max} \leq X_{max}$ where X_{max} is the maximum absolute value of a single coordinate in search space. For V_{max}=X_{max} particles can move outside of the defined search space borders. If a particle is inside the defined search space, then every position in it is inside the movement range over a single iteration $[\vec{0}, \vec{X_{max}}]$. If a particle leaves the defined

search space, it can return provided that the positions inside have better fitness from those outside.

A more liberal *velocity limit* $V_{max} > X_{max}$ can also provide good results when used with one of the techniques for velocity manipulation, e.g. *inertia* or *constriction* that are discussed in Subsection 3.5.1.

3.2.9 Update of particle position

Particle movement is simulated by adding the new velocity vector $\vec{v}^{t+1}$ of each particle p to its current position $\vec{p}^{t}$ in every algorithm iteration. The resulting vector is stored as the new particle position $\vec{p}^{t+1}$ as shown in Equation 3.4. The geometric interpretation of Equations 3.4 and 3.2 is given in Figure 3.2.

$$\vec{p}^{t+1} = \vec{p}^{t} + \vec{v}^{t+1} \tag{3.4}$$

3.2.9.1 Limiting particle position

In order to guarantee that particles stay withing the defined search space I^m, where $I = [X_{min}, X_{max}]$, each component p_i of the new particle position $\vec{p}^{t+1}$ should be limited as shown in Equation 3.5. If the dynamic *particle range* is not limited, inefficiency may result, since search could be conducted over a bigger space than necessary.

$$limit_p\,(p_i) = \begin{cases} p_i & \text{if } p_i \in [X_{min}, X_{max}] \\ X_{min} & \text{if } p_i < X_{min} \\ X_{max} & \text{if } p_i > X_{max} \end{cases} \tag{3.5}$$

An alternative approach is to stop any movement that would take the particle further away once it is already outside of the defined search space. This is achieved by setting the appropriate component v_i of the particle velocity vector $\vec{v}^{t+1}$ to zero. The exact procedure is shown in the following code fragment:

if $(p_j > X_{max}$ and $v_j > 0)$ or $(p_j < -X_{max}$ and $v_j < 0)$ **then**
  $v_j := 0$
end if.

After stopping for a few iterations, the particle will again move inside the defined search space, since it is drawn there by the group's *gbest* position. Only if the particle finds the global optimum outside of the defined search space, it will not return, because both its *hbest* and *gbest* will be set to the position of the global optimum. In that case the particle in question would lead a number of other particles outside of the defined search space. This problem can be prevented by

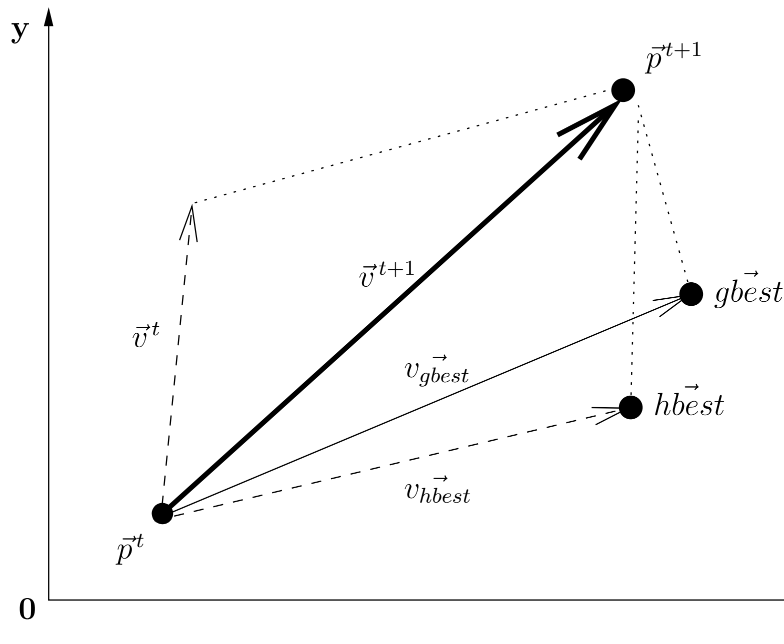

Figure 3.2: Geometric interpretation of update of particle velocity and position. The meaning of the variables is the following: vectors $\vec{p}^t$ and $\vec{v}^t$ are old particle position and velocity. Vectors $v_{\vec{hbest}}$ and $v_{\vec{gbest}}$ are the components of the new velocity vector $\vec{v}^{t+1}$, oriented at particle's $\vec{hbest}$ and $\vec{gbest}$ position respectively. The new particle velocity vector $\vec{v}^{t+1}$ is obtained by adding a linear combination $\Delta\vec{v}^{t+1}$ of $v_{\vec{hbest}}$ and $v_{\vec{gbest}}$ to the old velocity vector $\vec{v}^t$ according to Equation 3.2. The new particle position $\vec{p}^{t+1}$ is the sum of $\vec{p}^t$ and $\vec{v}^{t+1}$ as in Equation 3.4. [40]

modifying the *fitness function* to assign the worst fitness to all positions outside of the defined search space.

3.2.10 Values for the cognitive and social constant

The value of 2 is a commonly used, e.g. in the original paper on PSO [35], but not the only possible value for the *learning factors* in Equation 3.2. These parameters can vary from one PSO implementation to another. It is reported that values of less than 2, e.g. 1.49, are also not uncommon [41, 39].

No definitive conclusions have been reported on the case when the *learning factors* have asymmetric values [41]. In experiments in Chapter 5, Subsections 5.1.2.1, 5.1.3, 5.2.2, and 5.2.3, *asymmetric* and *symmetric* learning factors have both demonstrated optimal behavior in some cases. What is known, however, is that using only the *cognitive* or *social constant* alone does not bring good perfor-

mance [41]. Experiments in Chapter 5 have shown that PSO search does not work at all without the social constant, while only using the social constant works, but with suboptimal performance.

Slightly decreasing the value of the social constant and increasing the value of the cognitive constant should, in theory, emphasize independent particle search and deemphasize group search. Alternatively, group search can be emphasized and individual search deemphasized, by increasing the social constant and decreasing the cognitive constant.

The learning factors limit the maximum change of particle velocity, i.e. movement granularity in a single iteration step. The user can guide the behavior of the PSO simulation by controlling these parameters.

The quality of the cognitive and social constant is related to the specific problem that is being analyzed. Problem parameters include the size of the search space, the shape of the objective function used as the fitness evaluation function, the number of particles in the simulation, etc.

3.2.11 Limitations of the continuous algorithm

While the continuous PSO algorithm performs well on continuous objective functions, some research shows that it can have difficulties dealing with discrete variables [41]. In a discrete environment, the continuous PSO can degenerate to random search. The concept of velocity and position update inspired by movement in Euclidean space (Equation 3.2) may not apply to discrete spaces. The continuity of objective functions (*sphere*, *Griewank*, *Rastrigin*, *Rosenbrock*) employed as *test optimization problems* in the papers referred to in [41] could have been one reason for the apparent *stability* of the continuous PSO. Stability in this sense implies that the particle population does not change its mode of behavior with every change of the environment [42]. The prospect of instability of the continuous PSO for discrete objective functions motivates the introduction of discrete binary PSO, operating on a probability search space.

3.3 The Discrete Binary PSO Algorithm

The *discrete binary* PSO algorithm (DPSO) translates the idea of particle position and velocity update from the *Euclidean space*, used in the continuous PSO algorithm, to the *probability space*. Each bit that encodes a certain particle coordinate in the Euclidean space now becomes a vector component.

3.3.1 The probability search space

The DPSO algorithm operates on an $m \times n$ dimensional *probability search space* $\mathbb{P}^{m\times n}$ where m is the number of coordinates in the ordinary sense, and n is the number of bits used to encode each coordinate. The set $\mathbb{P}$ contains real values from the interval $J =]0,1[$. By using a simple stochastic experiment $\mathtt{s}$, probability vectors $\vec{p} \in \mathbb{P}^{m\times n}$ are converted to binary vectors $\vec{b} \in \mathbb{B}^{m\times n}$ where $\mathbb{B} = \{0,1\}$. Afterwards the vectors $\vec{b}$ can be decoded into coordinates of the Euclidean search space $\mathbb{X}^m$ by a certain decoding function $\mathtt{d}$. The set $\mathbb{X}$ contains numbers from some interval $I = [X_{min}, X_{max}]$; the size of I is determined by the number of encoding bits n. The conversion between different spaces is illustrated in Equation 3.6.

$$\mathbb{P}^{m\times n} \xrightarrow{\mathtt{s}} \mathbb{B}^{m\times n} \xrightarrow{\mathtt{d}} \mathbb{X}^m \tag{3.6}$$

3.3.2 The discrete binary particle

The *discrete binary particle* can be represented by an $m \times n$ matrix where m is the number of coordinates, i.e. the number of dimensions in the Euclidean search space, and n is the number of bits needed to encode a single coordinate.

3.3.2.1 Probability position of discrete binary particle

Each entry $p_{ij} \in]0.0, 1.0[$ of the $m \times n$ *matrix of bit probabilities* $\mathbf{P_p}$ contains the probability that bit j from dimension i will assume a zero or one. The whole matrix is a complete representation of the *particle probability position.* Each row encodes the probability of one coordinate in $\mathbb{X}^m$.

$$\mathbf{P_p} = \begin{pmatrix} p_{11} & .. & p_{1n} \\ & \ddots & \\ p_{m1} & .. & p_{mn} \end{pmatrix} \tag{3.7}$$

3.3.2.2 Realized position of discrete binary particle

Each entry $r_{ij} \in \{1,0\}$ of the $m \times n$ *matrix of realized bits* $\mathbf{P_r}$ contains the realized value of bit j from dimension i. This matrix is computed by applying the *stochastic experiment* in Equation 3.11 upon entries of the $\mathbf{P_p}$ matrix.

$$\mathbf{P_r} = \begin{pmatrix} r_{11} & .. & r_{1n} \\ & \ddots & \\ r_{m1} & .. & r_{mn} \end{pmatrix} \tag{3.8}$$

Matrix $\mathbf{P_r}$ contains the binary encoded realized position of a particle. It can be *decoded* as a vector $\vec{p}$ in Euclidean space by a decoding algorithm `d` (Subsection 3.4.5). Each component of $\vec{p}$ is decoded from one row of $\mathbf{P_r}$ (Equation 3.9).

$$\mathtt{d}(\mathbf{P_r}) = \vec{p} = (p_1, \ldots, p_m) \tag{3.9}$$

3.3.2.3 Probability velocity of discrete binary particle

Each entry v_{ij} of the $m \times n$ *matrix of probability velocity* $\mathbf{V_p}$ contains the value of probability change for each bit j from dimension i. This matrix is used to update the matrix of bit probabilities $\mathbf{P_p}$ (Subsection 3.3.7).

$$\mathbf{V_p} = \begin{pmatrix} v_{11} & .. & v_{1n} \\ & \ddots & \\ v_{m1} & .. & v_{mn} \end{pmatrix} \tag{3.10}$$

3.3.3 Algorithm pseudocode

Algorithm 2 shows the pseudocode of discrete binary PSO. Basic steps of the algorithm are: initialization of particles (memory, position, and velocity), calculation of fitness, update of bit probability velocity based on *gbest* and *hbest*, and the update of bit probability. Bit probabilities are *realized* to 0 or 1 values in binary space by a stochastic experiment. Afterwards, vectors from binary space are decoded to Euclidean space in order to apply the fitness function and evaluate each position.

Algorithm 2 Pseudocode of the discrete binary PSO algorithm

```
 1: Procedure PSO_DISCRETE ()
 2: Initialize particles
 3: while terminate() is false do
 4:    For each particle: Evaluate current position p⃗
 5:    Update neighborhood relation between particles
 6:    for each particle do
 7:       Determine if current position p⃗ better than hbest position
 8:       Determine gbest position in particle's neighbourhood
 9:       Update probability velocity Vp
10:       Update probability position Pp
11:       Derive realized position Pr from Pp
12:       Decode Pr as a vector p⃗ in Euclidean space
13:    end for
14: end while
```

3.3.4 Deriving of realized position from probability position

In line 7 of Algorithm 2, the realized particle position is obtained by applying a simple *stochastic experiment* on entries of the matrix of probability positions $\mathbf{P_p}$. The experiment consists of testing whether each entry $p_{ij} \in \mathbf{P_p}$ has a greater or equal value than a *random variable* $\mathbf{X}$ that is *uniformly distributed* over an interval $[0, 1]$. The result of the test in Equation 3.11 is encoded as a 0 or 1 entry in the matrix of realized positions $\mathbf{P_r}$.

$$p_{ij} \geq \mathbf{X}, \text{ where } \mathbf{X} \sim \mathbf{U}[0, 1] \tag{3.11}$$

The random variable $\mathbf{X}$ in the above test is usually implemented by a *random number generator*. The *seed* used for the generator should be different in each PSO run. Otherwise, two PSO runs with the same initial settings would perform an identical simulation and produce the same optimization results.

The pseudocode for the derivation of *realized particle position* is given in the following code fragment:

```
for i= 1 to m do
  for j= 1 to n do
    if ( p_ij ≥ rand()) then
      r_ij = 1
    else
      r_ij = 0
    end if
  end for
end for.
```

3.3.5 Decoding of realized position and calculation of fitness

In line 12 of Algorithm 2, the realized position of each particle `p` in binary space $\mathbb{B}^{m \times n}$ is decoded as a vector $\vec{p}$ in Euclidean space $\mathbb{X}^m$ according to Equation 3.9. A selection of methods used for encoding in binary space are discussed in Subsection 3.4.5.

Particle fitness is calculated in line 4 of Algorithm 2 by a *fitness function* f which evaluates the current particle position $\vec{p}$ in Euclidean space. The choice of the fitness function is problem dependent. A selection of fitness functions useful in the domain of worst-case execution time analysis is discussed in Subsection 4.3.2. The following code fragment demonstrates the position decoding and fitness calculation for some particle `p`. Variable m is the number of dimensions in Euclidean

space, while r_i is the i-th row of matrix $\mathbf{P_r}$.

for i = 1 to m **do**
  p_i = decode_binary_stream (r_i)
end for
fitness(p) = $f(\vec{p})$

3.3.6 Update of probability velocity

After the fitness of each particle is calculated, it is possible to infer the *hbest* position of each particle in line 7 of Algorithm 2. With the additional information of *neighborhood relation* between particles it is possible to determine the *gbest* position in line 8 of the same algorithm. The rules for the update of *hbest* and *gbest* are similar to those of the basic PSO algorithm. The new *probability velocity* of each particle is updated in line 9 based on the particle's *hbest* and *gbest* position. The pseudocode for the update of probability velocity is displayed in Algorithm 3.

Since the terms $(hbest_{ij} - r_{ij})$ and $(gbest_{ij} - r_{ij})$ in Algorithm 3 can only have three possible values from the set $\{-1, 0, 1\}$, the factors c_1 and c_2 together with the random factors $rand_1$ and $rand_2$ are dominant in determining the absolute change Δv_{ij} of each velocity component.

The result of the difference terms only controls the direction of Δv_{ij}. Again, m is the number of dimensions in Euclidean space, and n is the number of bits per dimension.

The factors c_1 and c_2 have the same role as the *cognitive* and *social constant* in the continuous PSO algorithm. Their exact value is a parameter of the system; often used values are $c_1 = c_2 \in [0.0, 2.0]$.

Algorithm 3 Velocity update in discrete binary PSO

for i = 1 to m **do**
  for j = 1 to n **do**
    $v_{ij} = v_{ij} + \overbrace{c_1 \cdot rand_1 \cdot (hbest_{ij} - r_{ij}) + c_2 \cdot rand_2 \cdot (gbest_{ij} - r_{ij})}^{\Delta v_{ij}}$
    $v_{ij} = limit_v(v_{ij})$
  end for
end for

3.3.7 Update of probability position

The pseudocode for the update of *probability position* in line 10 of Algorithm 2 is shown in the following code fragment:

for i = 1 to m **do**
  for j = 1 to n **do**
    $r_{ij} = S(v_{ij})$
  end for
end for.

The probability position update in discrete binary PSO is only a function of probability velocity and not a function of both probability velocity and previous probability position, which would be analogous to continuous PSO. The probability position is limited to the interval]0.0, 1.0[by the *sigmoid function* $S(x)$ [38].

$$S(x) = \frac{1}{1 + e^{-x}} \tag{3.12}$$

3.3.8 Maximum value of probability velocity

The function *limit_v* that limits the probability velocity in Algorithm 3 is the same as the function in Equation 3.3 used for the continuous PSO. It has the task to limit the entries of the $\mathbf{V_p}$ matrix to a scalar value V_{max}, which together with the learning factors c_1 and c_2 is another parameter of the DPSO system. The value of V_{max} controls whether new realized positions shall be tried even after the particle attains a certain probability position, i.e., the probability position that leads to certain realization of the currently known optimum in Euclidean space to which all the particles are converging.

Examples of absolute V_{max} values recommended in literature are 6.0 and 10.0 [38], whereas other values are also possible. For example, if V_{max} is set to 6.0, then according to Equation 3.13 there will be an approximate probability of 0.0025 that a bit entry in $\mathbf{P_r}$ may flip even when its corresponding component of probability velocity, v_{ij}, reaches V_{max} or $-V_{max}$. For V_{max}=10.0, the probability of a random bit flip is even smaller, only 0.000045.

Using the language of genetic algorithms: the value of V_{max} controls the mutation rate of the particle's matrix of realized bits; smaller V_{max} values allow higher mutation rates. From the PSO point of view: the value V_{max} sets the limit on search space exploration after the particle swarm converged on a certain known optimum position.

$$r_{ij} = \frac{1}{1 + e^{-v_{ij}}} \tag{3.13}$$

3.3.9 Statistic analysis of the update of probability velocity

The velocity update in discrete binary PSO 3 can be analyzed case by case for different values of $hbest_{ij}$, $gbest_{ij}$, and r_{ij}. Since each of the variables is binary,

Case	$hbest_{ij}$	$gbest_{ij}$	p_{ij}	Δv_{ij}
1.	0	1	0	$-\Psi_2$
2.	0	1	1	$+\Psi_1$
3.	1	0	0	$-\Psi_1$
4.	1	0	1	$+\Psi_2$
5.	1	1	0	$-(\Psi_1 + \Psi_2)$
6.	1	1	1	0
7.	0	0	0	0
8.	0	0	1	$+(\Psi_1 + \Psi_2)$

Table 3.1: Different cases of probability velocity update for DPSO.

eight different cases, shown in Table 3.1, can be distinguished.

$$v_{ij} = v_{ij} + \overbrace{(r_{ij} - hbest_{ij}) \cdot \Psi_1 + (r_{ij} - gbest_{ij}) \cdot \Psi_2}^{\Delta v_{ij}} \tag{3.14}$$

Equation 3.14 is equivalent to the velocity update in Algorithm 3, but it uses the distributional notation for the product of random factors $rand_1$, $rand_2$ and learning constants c_1, c_2. In this notation, Ψ_1 and Ψ_2 are terms with a uniform distribution $\mathbf{U}[0, c_1]$ and $\mathbf{U}[0, c_2]$ respectively. If, as it is usual, the learning constants have the same value $c = c_1 = c_2$, this can be written as $\Psi_1, \Psi_2 \sim \mathbf{U}[0, c]$.

The *probability density* of velocity change Δv_{ij} was analyzed in [43] for a single particle in one-dimensional space. Fixed values were used for the binary variables *hbest* and *gbest*, and a *Monte Carlo* simulation was performed for each of the four cases. The statistics collected during the simulation revealed a distribution of $50-50\%$ for a value of realized position $r_{ij} = 0$ and $r_{ij} = 1$. Based on this fact and the uniform distribution of Ψ_1 and Ψ_2, it was possible to analyze the distribution of the velocity component change Δv_{ij}. This analysis is summarized in Table 3.1.

For Case 1 or 2, Δv_{ij} has a distribution $[-\Psi_2, \Psi_1]$, which is centered around zero. For Case 3 or 4, the distribution $[-\Psi_1, \Psi_2]$ of Δv_{ij} is also centered around zero. In Cases 6 and 7 no change of velocity occurs. Only in Cases 5 and 8 there is significant velocity change. In Case 5 the distribution of Δv_{ij} is $(-\Psi_1 - \Psi_2) = -(\Psi_1 + \Psi_2)$. The sum of these two uniform distributions yields a triangular distribution whose center is less than zero. In Case 8 the distribution of Δv_{ij} is $(\Psi_1 + \Psi_2)$ which is a triangular distribution whose center is greater than zero.

The work [43] concludes that the probability of significant velocity change Δv_{ij} is fairly small in discrete binary PSO, since it occurs only in Cases 5 and 8.

3.4 Algorithm Modules

Both the continuous and the discrete discrete binary version of PSO have in common the concepts of *topology*, *metric*, and *neighborhood relation*. The encoding of particle positions is straightforward in the continuous algorithm, whereas in the discrete binary version there are at least two possibilities: *binary* and *gray* encoding.

Each of the above concepts can be modelled in different ways with a varying degree of complexity. There is more than one algorithm to do the job for each task, but the input and output generally have to conform to the requirements set by the rest of the algorithm. The implementation of each concept leaves much room for freedom and presents many design choices, e.g., which metric shall be used to measure the distances between particles, how shall it be implemented, what topology best captures group dynamics and information sharing between the particles, and how much should a single *bit flip* in binary space be able to change the position of a particle in Euclidean space.

In the practical implementation it was deemed best to implement the solutions for each concept inside a separate module. Since the computational complexity inside each module approaches the complexity of the PSO algorithm, this organization provides good encapsulation and reduces the overall complexity of the implementation.

3.4.1 Topology of the Particle Swarm

Topology determines the number of neighbors for each particle. For each particle, the h nearest particles are considered its neighbors. Which particles are more or less distant is determined by the metric. The value of h can be the same for all particles as in the *star* and *circle* topology, or it can be different, e.q. in the *wheel* topology.

The *neighborhood relation* between two particles can be depicted graphically using the *neighborhood graph*. In it, each particle is represetned by a single vertex, and two particles in a *neighborhood relation* are indicated by a connection between two vertices. Usually, particle swarm algorithms deal with a single swarm. In any case, the neighborhood graph of each swarm has to be connected. It must be possible to perform a walk starting from each vertex to any other vertex in the graph.

The property of graph's connectedness guarantees that the global best position, discovered by a certain number of particles, can eventually propagate itself as the *gbest* position of every single particle in the swarm. Thus, particles tend to converge on the position of the global best particle, if such a position does not change for a sufficient number of iterations.

The choice of topology determines the propagation speed (number of iterations) of the global best position around the swarm. The more neighborhood relations between particles or connections in the neighborhood graph exist, the faster is the propagation of the currently optimal global best position.

3.4.1.1 Star topology

In the case of *star topology*, depicted in Figure 3.3, each particle stands in a neighborhood relation to every other particle. This topology is inspired by social groups where decisions are based on the consensus of all group members.

The *gbest* position of each particle is always the current, global best position of the entire swarm. The current global best position is propagated to all particles in a single iteration.

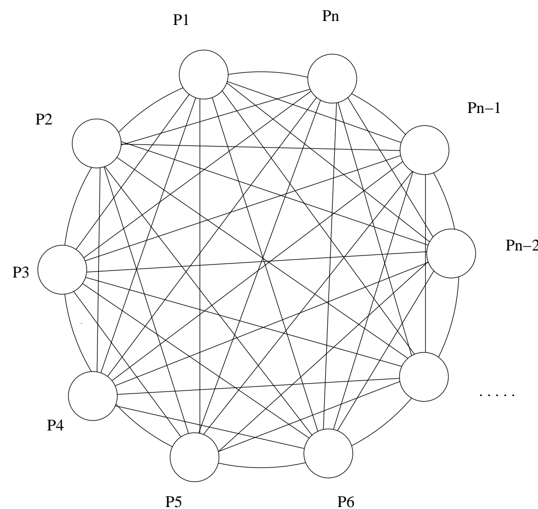

Figure 3.3: Particle swarm with a star topology

3.4.1.2 Circle topology

In the *circle topology* (Figure 3.4) each particle of the swarm has exactly k neighbors; distant particles are not necessarily standing in a neighborhood relation. The global best position of the swarm will propagate itself to all particles in at most $|S|/k$ iterations, where $|S|$ is the size of the swarm. A real circle results from this topology for $k = 2$.

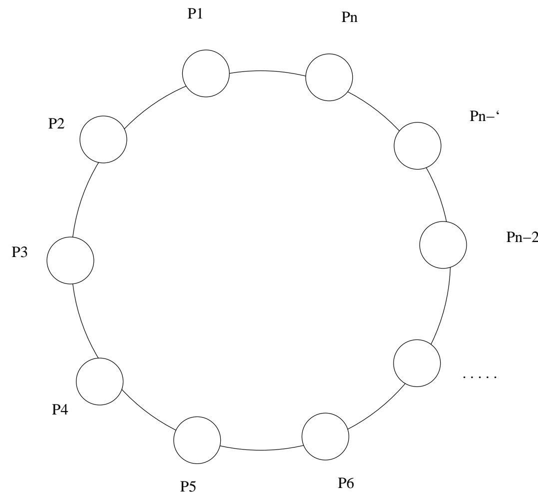

Figure 3.4: Particle swarm with a circle topology, for k=2

3.4.1.3 Wheel topology

The *wheel topology* (Figure 3.5) models hierarchical social groups where each group has a single leader. In PSO, the leader role is assumed by a single *focal* particle that is connected to all other particles on the periphery. Peripheral particles have no connections in between them. The propagation of the global best position takes at most two iterations for each particle on the periphery and a single iteration for the focal particle.

3.4.2 Distance metric

A *metric* determines the distance between two particles in search space. In the computational sense this creates a *total preorder* (TPO) of distances d_j for each particle p to all other particles p_i with $p \neq p_i$. The distances are ordered starting with the shorter distances first; in the case of ties, distance to the particle with lower index gets precedence. For some particle p, the total preorder of distances to other particles $p_1, \ldots, p_n$ is given as:

$$TPO(p): \ d_{i_1} \leq d_{i_2} \leq d_{i_3} \leq ... \leq d_{i_n}, \ where \ d_{i_j} = \delta(p, p_{i_j}) \ and \ i_j \in [1, n] \quad (3.15)$$

Depending on the topology used, only a few starting elements of the TPO of each particle may be necessary for building the neighborhood relation. For the star topology, no TPO is needed, since all particles are neighbors. For the wheel topology, the TPO is also not needed, since a single focal particle stands in the neighborhood relation to all other particles regardless of distance. However, the

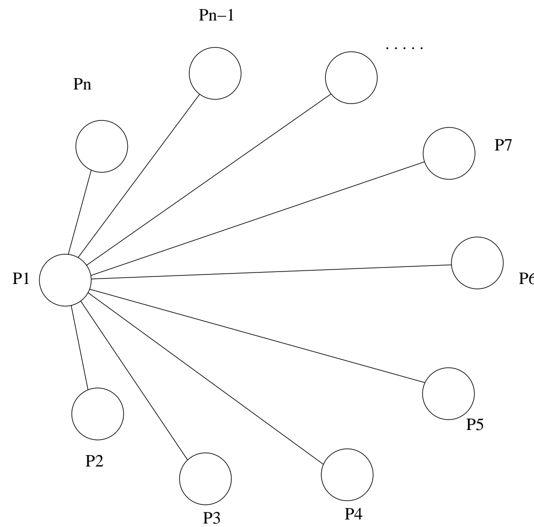

Figure 3.5: Particle swarm with a wheel topology

distance preorder is relevant for the circle topology. Although the circle topology specifies that each particle has k neighbors, it does not state which of the $|S| - 1$ possible particles should be considered as such. When using this topology, only the first k particles of the TPO are defined as neighbors.

The following subsections present a selection of metric definitions that can be used in PSO.

3.4.2.1 Euclidean metric

The distance between positions $\vec{p}$, $\vec{q}$ of two particles p, q in continuous PSO can be calculated using the *Euclidean metric* as:

$$\delta(p, q) = ||\vec{p}, \vec{q}|| = \sqrt{\sum_{i=1}^{m} (p_i - q_i)^2}, \tag{3.16}$$

where m is the number of search space dimensions.

3.4.2.2 Probability based Manhattan metric

The distance between two particles in discrete binary PSO can be defined using the *Manhattan* or *Taxi metric*, as the sum of absolute values of bit probability differences. Using the notation for the discrete binary particle given in Subsec-

tion 3.3.2.1, this can be expressed as:

$$\delta(\mathbf{P_p}, \mathbf{Q_p}) = \sum_{i=1}^{m} \sum_{j=1}^{n} |p_{ij} - q_{ij}|, \tag{3.17}$$

where m is the number of dimensions in the ordinary sense and n the number of bits needed to encode a single coordinate.

3.4.2.3 Metric based on fitness

It is possible to define the distance between two particles based on the absolute difference of their respective fitness values. For two particles p and q the distance can be calculated as:

$$\delta(p, q) = |fitness_function(\vec{p}) - fitness_function(\vec{q})| \tag{3.18}$$

.

The advantages of the *fitness based metric* are its relative simplicity and low computational requirements. It is generic and can be applied to both the continuous and the discrete binary version of PSO. However, it is suspected that such a metric might cause premature convergence of particles on the current global best position, since all particles with the highest fitness would be standing in the neighborhood relation. If this is the case, the chance of particles getting stuck at a local optimum would be increased.

The motivation for using the fitness based metric is the following. Particles with high fitness values would be in the same neighborhood, and they would tend to concentrate their search in the area of search space around the current optimum. Thus, they would perform *local search* around the current optimum and fine-tune this solution. Other particles with mutually similar fitness would also build neighborhoods. However, they would be attracted to the the neighborhood of particles with the highest fitness. From this follows that particles with lower fitness values would conduct a more global search, since according to the velocity update in Equation 3.2, their velocities should be greater than those of the particles already in the optimum area of search space. The reaction of particle neighborhoods with lower fitness values to changes of the current, global optimum position would be limited by the rate of *gbest* propagation, which depends on the used topology.

Experiments in Chapter 5 showed that the fitness based metric is not inferior to other proposed metrics; in some cases it even outperformed them in terms of achieved fitness.

3.4.3 Neighborhood relation

A *neighborhood relation* is a prerequisite for determining the *gbest* position of each particle. Generally, the *gbest* position of some particle p can be determined using the following procedure:

```
1: double gbest_fitness = p → fitness;
2: (p → gbest) := p;
3: for q in Neigh (p, k) do
4:    if (q → fitness) > gbest_fitness then
5:       gbest_fitness := (q → fitness);
6:       (p → gbest) := q;
7:    end if
8: end for.
```

Function $Neigh(p, k)$ gives a list of particles in order of their distance to p. Parameter k is the number of particle's neighbors, which depends on the topology definition.

For the star and wheel topologies the neighborhood relation is statically fixed at the beginning of execution and needs not be calculated at run-time. Thus, the list of neighboring particles is always constant.

For the circle topology the neighborhood relation can change at run-time. The reason for this is, that the distance between particles, which naturally changes as particles fly in the search space, determines what k particles are the closest and should be selected as neighbors of the currently observed particle. The *gbest* position in every iteration is not selected from a fixed list of particles as in the star and wheel topology, but from a dynamically changing list.

As previously stated, the neighborhood relation in the star and wheel topology is determined without any need for a metric. For the circle topology, the neighborhood relation can be constructed both with and without the metric definition.

1. If there is no metric available, it is possible to define k neighbors of the observed particle based on their respective position in the particle array. For $k = 2$, the neighbors of particle p_i would be particles p_{i-1} and p_{i+1}.

2. If there is a metric available, the neighborhood relation between particles can be modelled as an $n \times k$ matrix $\mathbf{N}$, where each row i contains the list of neighbors of some particle p_i, sorted in the order of distances. The ordering of distances in the matrix can be carried out by a modified sorting algorithm.

$$\mathbf{N} = \begin{pmatrix} p_{11} & \cdots & p_{1k} \\ & \ddots & \\ p_{n1} & \cdots & p_{nk} \end{pmatrix} \tag{3.19}$$

The metric definition should be based on some relatively stable property of PSO particles. Abrupt changes in particle distances, i.e. the ensuing change of the neighborhood relation can cause *instability* of the algorithm (Subsection 3.2.11). The neighborhood relation determines the set of possible *gbest* positions for each particle. The lack of sufficient permanence of one *gbest* position for a certain group of particles can prevent them from converging to some *gbest* position, i.e., from exploring the region of space around it.

An example of a stable property in continuous PSO, suitable for a metric definition, would be the particle position in Euclidean space. For the discrete binary PSO, the stable properties would be the particle position in probability space. The realized particle position in binary space would not count as a stable property for DPSO, since its change can be more abrupt than the corresponding change of the probability position. This is a consequence of the random method whereby the realized position is derived from the probability position.

Whether the property of particle fitness can be used to create a stable metric definition for discrete binary PSO is not easy to answer. Since particle fitness is a function of realized particle position, it follows that the fitness can change as often as the position. How often the fitness, and thus, the neighborhood relation will change significantly, depends on the definition of the fitness function. While the neighborhood relation based on a *fitness metric* can be stable for some fitness functions, it can also be unstable for others. How a fitness based metric, i.e. the neighborhood relation that results from it and the circle topology, performs in practice is explored in Chapter 5

Figure 3.6 shows different neighborhood relations for the *circle topology* that ensue from different metric definitions.

3.4.4 Effect of topology on performance

It was shown in [44] that there are statistically significant differences in performance between the *circle* and *wheel topology* on certain benchmark fitness functions. However, the same work also concluded that, although the circle and wheel topology were useful in combination with some fitness functions, for the majority of benchmarks, the best results were provided by the star and random topology.

It is widely accepted that topology affects the speed of information flow of *gbest* between particles. Both [44] and [45] investigated the *star*, *circle*, and *wheel topology* on a set of standard benchmark functions. However, no significant influence of topology on the convergence behavior of particles was detected.

In order to control particle convergence, especially premature convergence, a concept of *particle bouncing* was proposed in [45]; this topic is discussed in Subsection 3.5.2. The Clerc's constriction factor (Subsection 3.5.1.2) or an appropriate setting for V_{max} (Subsection 3.2.8.1) can also prevent premature conver-

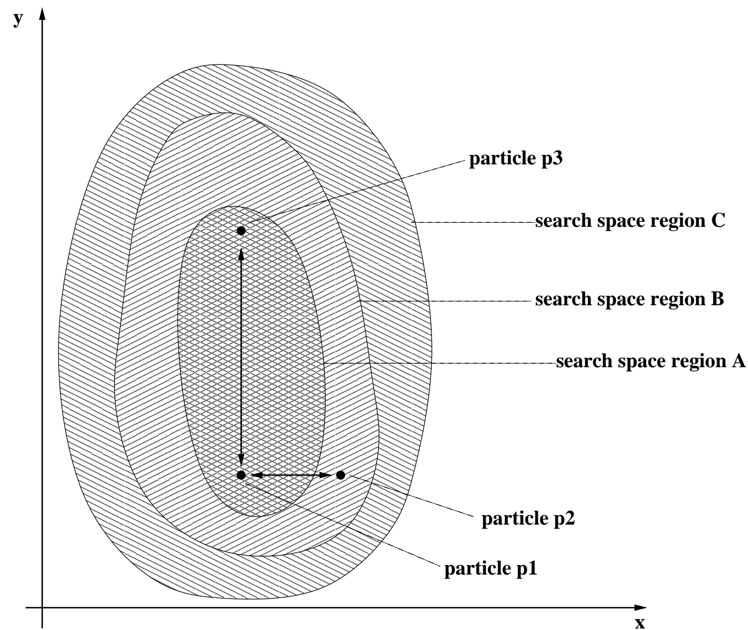

Figure 3.6: Depending on whether a metric based on fitness or on particle coordinates is used, there can be a difference which of the particles p_2, p_3 is considered as the first neighbor of particle p_1. Using a metric based on coordinates (Euclidean, Manhattan), particle p_2 would be considered to be nearest to particle p_1, i.e., it would be its first neighbor. Using a fitness based metric would result in particle p_3 being the first neighbor of particle p_1, since both particles are inside the region of search space with the same fitness. The change in distances between particles causes a change in their neighborhood relation when the particle swarm uses a circle topology. This influences the calculation of g_{best} position for each particle.

gence. However, the correct V_{max} setting is problem dependent, while *bouncing* and *constriction* seem to be more generic approaches. The above observations were derived from experiments performed on continuous PSO using continuous benchmarks functions.

In the overview of PSO [41], it is stated: 'The effects of different neighborhood structures on performance are inconclusive'. However, it is not clear from that and the above mentioned works what kind of a metric definition was used in constructing the *neighborhood structures*. In particular for the *circle topology*, it would seem that no consideration was given to the metric definition as it was done in Subsection 3.4.3 of this work.

Furthermore, the effect of different topologies on discrete binary PSO was not explored neither in the original paper [38] nor in the probability analysis given

in [43]; it seems that only the star topology was used. The evaluation of other topologies is carried out in Chapter 5 of this bachelor thesis.

3.4.5 Encoding of position coordinates

The *encoding function* `e` is the inverse of the decoding function `d` defined in Subsection 3.3.1. In DPSO, function `e` is used to convert the initial positions of particles from Euclidean space $\mathbb{X}^m$ to binary space $\mathbb{B}^{m\times n}$. The initial positions in space $\mathbb{B}^{m\times n}$ can be used to derive the initial positions in probability space $\mathbb{P}^{m\times n}$. The *decoding function* `d`, on the other hand, is needed to prepare the positions in $\mathbb{B}^{m\times n}$ for evaluation by the fitness function in $\mathbb{X}^m$. This task occurs in every iteration of DPSO.

In principle, any data type used as an argument for the *fitness function* can be encoded using a binary representation. The implementation of DPSO presented in Chapter 4 supports *int* and *double* data types. The int type can be encoded using ordinary *binary* or *gray encoding.* The double type is based on the encoding of the int type; the fractional part is obtained by dividing the int with 10^d; constant d determines the precision i.e. the number of decimal places.

3.4.5.1 Binary code

If ordinary *binary code* is used for encoding, then each coordinate p_i of $\mathbb{X}^m$ can be decoded from $\mathbb{B}^n$ by the following equation:

$$p_i = -1^{b_n}(\sum_{j=0}^{n-1} 2^j \cdot b_j), \text{ where } b_j \in \{0,1\}. \tag{3.20}$$

Variable m is the number of dimensions in Euclidean space, and n is the number of bits needed to encode a single coordinate. The number of dimensions in binary space is $m \times n$.

When using the ordinary binary code, a single bit-flip can change the value of one coordinate by any power of 2, ranging from 2^0 to 2^n. With this encoding, some bits carry more weight than others, which might hurt the stability of DPSO.

3.4.5.2 Gray code

Gray code has the advantage over ordinary binary code in that the distance $\|x-y\|$ of two whole numbers x and y is equal to the *Hamming distance* between their respective bit representations. This property is thought to be useful in discrete binary PSO, since a single bit flip would always change the value of a particle coordinate by one. The only exception to this are the beginning and end of the

Bits	Number
000	-4
001	-3
011	-2
010	-1
110	0
111	1
101	2
100	3

Table 3.2: Encoding of the interval of whole numbers [-4,3] using Gray code.

encoded interval, which also have the Hamming distance of 1, but whose absolute difference is equal to the size of the encoded. An example of this can be seen in Table 3.2 for values -4 and 3.

The Gray code has previously been successfully used with another class of heuristic algorithms, namely the genetic algorithms [46]. It is hoped that it would likewise prove useful in discrete binary PSO by bringing more stability. Experiments in Sections 5.1.1 and 5.2.1 have confirmed this assumption to a high degree.

3.5 Extensions of the PSO Algorithm

Since heuristic algorithms are not an exact science, there exist many possibilities for expanding the concepts of simple PSO presented in papers [35, 38]. Each of the concepts can be extended with new rules and an increased level of sophistication in *particle decision making*. However, while some of the extensions may prove beneficial (in some target domains), others will be slashed with the Occam's razor [1]. In principle, these extensions can be implemented with both the *continuous* and the *discrete* PSO algorithm. In practice, the results between the two implementations can vary significantly. What seems to be a good extension of continuous PSO does not necessarily provide good results with the discrete version.

3.5.1 Manipulation of particle velocity

The formula for *velocity update* in Equation 3.2 is a central piece of the PSO algorithm and a good place to start with modifications. It can be extended to model concepts such as *inertia* and *constriction*. Each of these modifications

[1]Occam's Razor - A principle attributed to the 14th century logician and Franciscan friar William of Ockham stating that 'entities should not be multiplied unnecessarily'.

tries to balance *global* and *local search* by influencing particle velocities. Higher velocities put more emphasis on global search, while lower velocities emphasize local search. Usually, it is desirable to emphasize global search at the beginning and local search towards the end of the PSO run. Local search is thus limited to exploration of those regions of search space which were found to contain the best candidates for the optimum during the preceding phase of global search.

3.5.1.1 Inertia

A simple extension to the velocity update formula is to model *inertia*, which is a characteristic of matter to stay in its current state of motion; energy is necessary to bring the matter out of its current state.

The parameter for *inertia weight* w in Equation 3.21 models the influence of old particle velocity $\vec{v}^t$ upon the new velocity $\vec{v}^{t+1}$. If inertia weight is greater than 1, then particle velocity from the previous iteration has more influence on the new velocity. If inertia is less than 1, old particle velocity is proportionally less represented in the calculation of new velocity.

$$\vec{v}^{t+1} = w \cdot \vec{v}^t + c_1 \cdot rand_1 \cdot (\vec{hbest} - \vec{p}^t) + c_2 \cdot rand_2 \cdot (\vec{gbest} - \vec{p}^t) \tag{3.21}$$

In [47], it was found that the inertia parameter $w = 0.7$ is optimal for the fitness functions investigated by the authors. Furthermore, it was found in [37] that a linearly time decreasing inertia weight, starting from 0.9 and ending at 0.4, discovers the optimum in even more cases. However, the difficulty is that the rate of linear decrease has to be manually set. Furthermore, the obtained values for w may only be optimal for the fitness function and the exact experiment settings used by the authors. The reported experiment was carried out using the continuous PSO algorithm.

3.5.1.2 Constriction

A different approach for balancing between global swarm 'exploration' and local 'exploitation' is to insert a constriction factor χ in the formula for velocity update, as shown in Equation 3.22.

$$\vec{v}^{t+1} = \chi[\vec{v}^t + c_1 \cdot rand_1 \cdot (\vec{hbest} - \vec{p}^t) + c_2 \cdot rand_2 \cdot (\vec{gbest} - \vec{p}^t)] \tag{3.22}$$

According to [36], the constriction factor χ is calculated using the following equation:

$$\chi = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}, \text{ where } \phi = c_1 + c_2,\ \phi > 4.. \tag{3.23}$$

As in ordinary velocity update, factors c_1 and c_2 stand for the *cognitive* and *social constant*

It has been reported in [36], that using the constriction factor χ guarantees stability and the convergence of particles on optimal regions of search space. Furthermore, in order to obtain the best results the author has recommended setting the maximum dimensional velocity, V_{max}, to the maximum possible value of a single coordinate X_{max}.

In [39] it has been reported, that the velocity update using constriction χ in Equation 3.22 is a special case of the velocity update using inertia w in Equation 3.21, if w is set to χ, and c_1 and c_2 meet the conditions in Equation 3.23. Authors of the above article have concluded that both inertia and constriction can provide good performance, whereas the advantage of constriction is that it does not require fine tuning. With inertia, fine tuning of parameters w, c_1, and c_2 can bring an additional increase in performance. The reported results were obtained using the continuous PSO algorithm with a set of standard benchmark functions for evolutionary algorithms, namely the *spherical*, *Rosenbrock*, *Rastrigin*, *Griewank*, and *Schaffer's f6* function.

3.5.2 Spatial extension of particles

The *spatial extension* of particles is an attempt to avoid the crowding of particles at a single position in search space by introducing *particle volume*, that is, an area of search space around the particle, inside which there can be no other particles. Usually, the volume is defined by assigning to each particle a certain radius r which is a parameter of the system.

One way of achieving the requirement that there are no particles y around a certain particle x with radius r is by *particle bouncing*. Bouncing consists of two parts: *collision detection* and *collision avoidance*.

3.5.2.1 Collision detection

In each iteration of the PSO algorithm, before particle position is updated by Equation 3.4, an additional step called *collision detection* is introduced. It checks whether moving a particle x with position $\vec{x}$ by the current particle velocity vector $\vec{v}$ would bring it on a collision course with some other particle y with position $\vec{y}$. Thus, if some particle y at time $t+1$ would have a position $\vec{y}^{t+1}$ that is inside the hypersphere around the future position $\vec{x}^{t+1}$ of particle x, a collision avoidance strategy needs to be applied to keep the two particles colliding. This condition is formalized in Equation 3.24. Variable m is the number of search space dimensions, and r is the radius of hypersphere R^m around the position of particle x at time

$t + 1$.

$$\vec{y}^{t+1} \in R^m(\vec{x}^{t+1}, r) \tag{3.24}$$

3.5.2.2 Collision avoidance

The proposed strategies for the avoidance of *particle collision* include: *random bouncing*, realistic *physical bouncing*, and simple *velocity-line bouncing*. It was reported in [45] based on experiments on continuous PSO that simple velocity-line bouncing provides the best performance. It was implemented in the following way:

When an upcoming particle collision is detected using the condition from Equation 3.24, the new velocity vector $\vec{v}^{t+1}$ is scaled by the *bounce factor* λ according to Equation 3.25. The bounce factor is a parameter of the system, and in principle, it can either be a constant or a random value. Furthermore, λ only scales the magnitude of particle velocity but does not change its direction.

$$\vec{v}^{t+1} := \lambda \cdot \vec{v}^{t+1} \tag{3.25}$$

Based on different values for λ, the following cases of collision avoidance can be distinguished:

1. For $\lambda \in [0.0, 1.0]$, the particle is slowed down.
2. For $\lambda > 1.0$, the particle is speeded up.
3. For $\lambda < 0$, the particle makes a U-turn.

The third case when particles make a U-turn is illustrated in Figure 3.7 for two dimensions.

3.5.2.3 Effect of Particle Bouncing

According to the authors in [45], *particle bouncing* facilitated by collision detection and avoidance provides a way for some particles to escape regions with a global or local optimum, while letting other particles stay and perform local search. Particle bouncing has been reported to prevent the *premature convergence* on local optima, after which the performace of PSO usually ‘flattens out‘.

During practical implementation, the author of this bachelor work has observed that premature convergence in discrete binary PSO can also be prevented by appropriate settings of velocity limit, V_{max}, and learning factors c_1 and c_2. Greater V_{max} values facilitate greater particle velocities, which in effect lead to faster convergence. Premature convergence can thus be controlled by reducing the value of V_{max}, so that particles converge only after a certain number of iterations has been reached. The exact value for V_{max} can be fine-tuned depending on the:

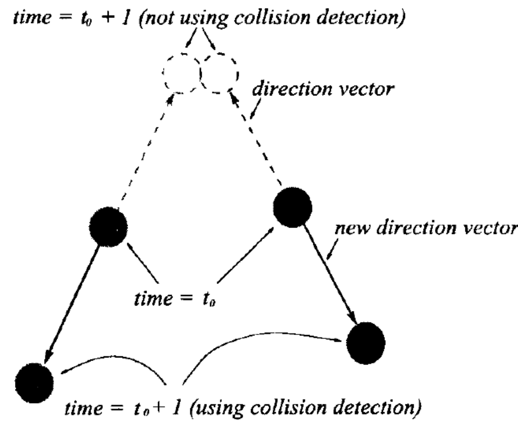

Figure 3.7: Particles making a U-turn with a bounce factor $\lambda = -1$. [45]

1. problem under optimization
2. computational resources and time available for each PSO run.

3.6 Summary

In this chapter, two versions of PSO were presented: the *continuous* and *discrete binary* PSO algorithm.

The discussed topologies, metrices, and the neighborhood relation are commonly applicable to both versions of the algorithm. However, the effect on performance may differ between the two versions.

The update of *particle velocity* and *position* differs significantly between the two algorithm versions. This is the consequence of different spaces in which the algorithms perform the search. The continuous algorithm operates in ordinary *Euclidean space*, while the discrete version operates in *probability space*. The velocity and position update in discrete binary PSO is an extension of the movement equation from continuous PSO.

The *evaluation* of particle positions is the same for both algorithms, and it operates on values from the Euclidean space. For this purpose, the discrete binary version of the algorithm has to convert values from probability to Euclidean space before fitness evaluation can take place. The conversion is achieved by a simple stochastic experiment.

In the discrete binary version of PSO, the choice of *encoding* determines the effect of single bit flips on the encoded particle position in Euclidean space. In contrast to ordinary binary code, Gray code guarantees that a single bit flip will

only change the value of a single coordinate in Euclidean space by one. This property can be beneficial for algorithm *stability*.

The settings of the parameters for velocity and position update: V_{max}, c_1, c_2, and X_{max} significantly influence the performance of PSO. Different velocity manipulation techniques can be used to find a balance between *global* and *local search* of the search space. Examples of these techniques are inertia and constriction.

Premature convergence on a local optimum can have negative influence on performance. It can be prevented with a spatial extension of particles, which provides a mechanism for *bouncing* of converged particles.

It is estimated that each of the two PSO versions has different performance in their respective target domains. The exact comparison of the two algorithms is performed in Chapter 5.

Chapter 4

Adaptation of PSO for Estimating WCET

This chapter goes into detail how PSO is applied to estimate *worst-case execution time* of software components and tackle the related problems of *execution path analysis*. The *pso_wcet*[1] framework implements the algorithms for continuous (CPSO) and discrete binary (DPSO) particle swarm optimization. It also contains the implementation of a brute-force random search algorithm. The results of random search are used as a control group of the two PSO implementations. The structure of the framework does not depend on the implemented optimization algorithms. In principle, any optimization algorithm could have been used for the optimizer part of the framework. Apart from estimating WCET and dealing with execution path analysis, the framework benchmarks the performance of PSO in this specific target domain.

4.1 Services of the PSO WCET Framework

The services provided by the *pso_wcet* framework are: preparing the *software-under-test* (SUT) for execution, iterative execution of the SUT, fitness calculation, and the generation of input data vectors. These services are graphically depicted as actions of the data-flow diagram in Figure 4.1.

[1] *pso_wcet* - The source code of this framework is publicly available under the simplified BSD license at `http://psowcet.sourceforge.net/`

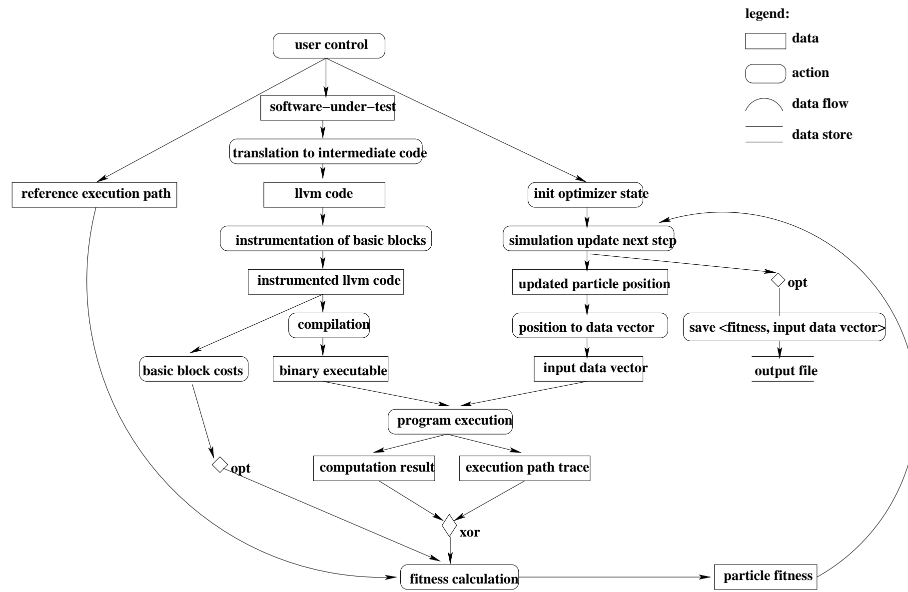

Figure 4.1: Data-flow diagram of the *pso_wcet* framework - The diagram consists of linear data-flow in the left, and cyclic data-flow in the right and lower middle part. The linear data-flow is a one time affair for each SUT. Its tasks are the instrumentation of intermediate SUT code and, optionally, the derivation of time costs of basic blocks. Additionally, the user can specify a reference execution path that is used as a target by some fitness functions. The cyclic part of the data-flow is iteratively repeated until the termination condition is reached. The three main phases of the cyclic part are: generation of input data vectors, SUT execution, and fitness calculation. The generated input data vectors can be stored to a result file for later reference.

4.1.1 Preparing the software under test

Preparing the SUT for execution includes translating it to intermediate LLVM[2] code, instrumenting the basic blocks (optionally deriving the execution-time cost of each block), and obtaining the user-specified reference execution path. The cost of each basic block is needed to estimate the WCET of the whole SUT. The user-specified reference path can serve as a target to generate input data that exercise it. Both *path search* and *execution time maximization* (WCET) can serve as goals for the fitness function.

[2]LLVM Project - A collection of modular and reusable compiler and toolchain technologies, available at http://llvm.org/

4.1.2 The testing cycle

The cyclic part of the framework consists of the following activities: generation of input data, SUT execution, and fitness calculation.

In each iteration the SUT is executed with new input data whose fitness is evaluated after the execution has terminated. The input data are a vector obtained from the position of a PSO particle in m-dimensional space where m is the number of parameters of the SUT main function.

The fitness functions can take the exercised execution path, the user specified reference path, and the costs of basic blocks as input. The fitness function for *path-length* takes only the exercised execution path. The function for *path-cost* also needs the costs of basic blocks. The functions for *path-similarity* - these are the prefix, set, and multiset path comparisons - need a *reference path* with which to compare each exercised path.

An additional option used for testing is that PSO can be applied on the result of computation of some SUT. This is used as a benchmark of the optimizer, since in literature [44, 45] various implementations of PSO have been tested on mathematical benchmark functions. Currently implemented are the following benchmarks: *sphere*, *Griewank*, *Rosenbrock*, and the *Rastrigin* function.

In the current implementation, the termination condition of the testing cycle is specified by the user; it can be a number of algorithm iterations or a goal specifying the number of discovered input data with certain fitness.

4.1.3 Saving the results of optimization

Depending on the mode of operation, the framework can save different fitness values and the corresponding input vectors to a result file. The following modes of result saving are implemented:

1. The framework saves the best input-vector of each iteration.

2. The framework saves only the best input-vector of the entire optimization run.

3. The framework saves all input vectors that fulfill certain fitness requirements. These requirements can be specified by the user as fitness intervals.

4.2 Software Under Test

In the current implementation, the software-under-test is a piece of C code inside a separate translation unit. It is characterized by a single main function that

needs to conform to one of the function signatures required by the optimizer. All functions in the same unit that are called directly from the main function or indirectly through other functions are considered to be part of the SUT. An exemplary SUT that consists of a single function is the implementation of the *bubble sort* algorithm shown in Listing 4.1.

Listing 4.1: Exemplary SUT - Bubble sort

```
/* file: bubble_sort.c */
double bubble_sort_i (int n, int array[]) {
  int i, j;
  for (i = 0; i < n; i ++) {
    for (j = 0; j < n - 1 ; j ++) {
      if (array [j] > array [j + 1]) {
        int tmp = array [j + 1];
        array [j + 1] = array [j];
        array [j] = tmp;
      }
    }
  }
  return 0;
}
```

4.2.1 Main function signature

The main function of the SUT has to conform to one of the function signatures given in Listing 4.2 in order to be compatible with the PSO optimizer. The *array* parameter in the function signature is an *input vector* which is used by the PSO optimizer to feed test data into the SUT. The array can be of variable length specified by parameter *n*. In the current implementation *int* and *double* arrays are supported.

Listing 4.2: Supported signatures of the SUT main function

```
double software_under_test_main (int n, int array[]);
double software_under_test_main (int n, double array[]);
```

4.2.2 Intermediate representation

The C code of an SUT is translated into an intermediate representation by using the clang-cc[3] compiler. The result of the translation is program code in LLVM

[3]clang-cc - A C, C++, Objective C and Objective C++ front-end for LLVM, available at http://clang.llvm.org/

virtual assembly language. In the output of clang-cc, each label starts a new basic block, which again corresponds to a node in the CFG of the software-under-test. The CFG and LLVM assembly code of *bubble sort* are shown in Figure 4.2.

4.2.3 Instrumentation

The basic blocks are instrumented by adding instrumentation code after each label in the LLVM code. The instrumented LLVM code is afterwards compiled into a binary executable. When the SUT is executed, the instrumentation code of each basic block fires upon execution reaching that basic block. This generates an *execution trace* of traversed basic blocks, which is used by the particle swarm optimizer for subsequent WCET and execution path analyses.

In the current implementation the instrumentation of intermediate SUT code is attained by using a script written in AWK[4]. The script searches for a pattern belonging to the start of an LLVM basic block and inserts *instrumentation code* at that location. The instrumentation code consists of a function call to a defined *callback function*. The callback from each basic block uses a unique ID which identifies the exercised basic block to the optimizer. The IDs are stored, and subsequently, the whole execution path can be reconstructed based on this information.

4.3 Particle Swarm Optimizer

4.3.1 Algorithms

The optimizer module in the current implementation contains three optimization algorithms: basic continuous PSO, discrete binary PSO, and random search. Random search was implemented in order to provide a control group for experiments with the two PSO implementations.

4.3.2 Fitness functions

Each of the implemented optimization algorithms uses a fitness function to evaluate a certain particle position, i.e., an input vector for the SUT. The available fitness functions can be divided into two groups.

The first group calculates fitness by evaluating the *execution path trace* generated by the SUT for a certain input vector. It can furthermore be subdivided into fitness functions which solely need the path trace for calculation and those which also need a certain *reference path* for comparison.

[4]AWK - A Turing-complete programming language designed for processing text-based data, available at http://www.gnu.org/software/gawk/

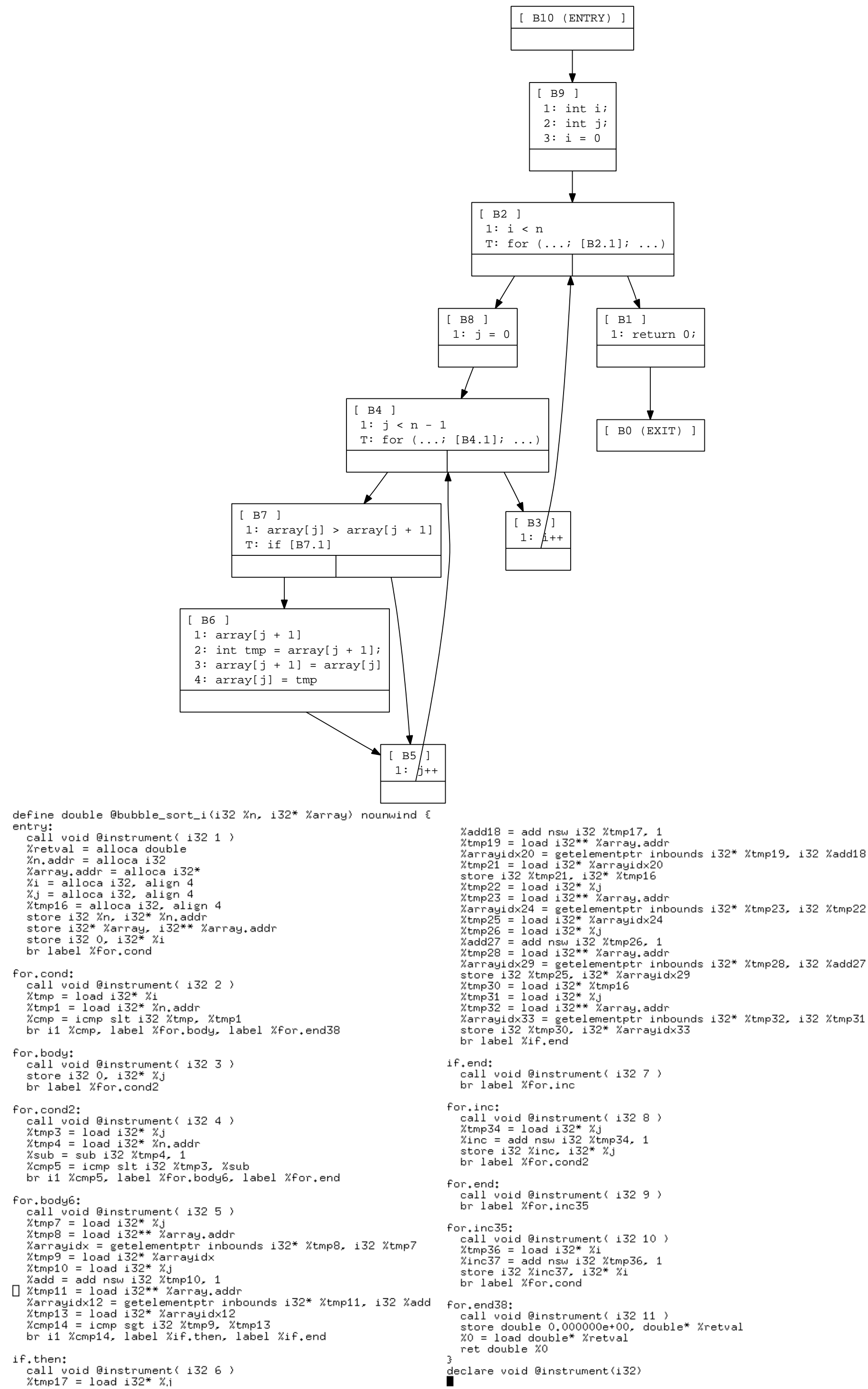

Figure 4.2: Instrumented LLVM code of SUT `bubble sort` from Listing 4.1 and the automatically generated CFG

Fitness functions which only need the path trace are the *path length* and the *path cost fitness functions*. Those which additionally need a reference path are the *path similarity fitness functions*.

If a fitness function does not need a path trace at all, it belongs to the second group. Such fitness functions work on the *computation result* of SUT and are used for function optimization in the classic sense. Each of the above fitness functions can be used for minimization or maximization.

4.3.2.1 Path length

In the case of fitness based on *path-length*, the total number of basic blocks in a path is taken as the fitness of the input vector, i.e., the particle position which induced the path.

4.3.2.2 Path cost

The fitness function based on *execution path cost*, calculates fitness by summing up the cost of each basic block in the execution trace. The time cost of each basic block has to be provided beforehand by the user. The *unit of execution time* can be defined as one processor cycle or some other unit of time, e.g., μs, *ns*, etc.

If the *pso_wcet* tool, presented in this thesis, would be applied for WCET estimation on a computer system free from timing anomalies, it would be necessary to provide realistic *instruction execution times*. Since currently a virtual machine with LLVM instructions is used, it was deemed best to give each instruction the time cost of one. The focus of this work is less on measurement of instruction execution times and more on the analysis of execution paths and the generation of input data. The input data generated using *pso_wcet* can among other applications be used for measurements of single instruction execution times by using some measurement framework to perform the actual, local, low-level timing analysis.

4.3.2.3 Path similarity

Fitness functions based on *path similarity* compare the path generated by an input data vector with some reference path. Usually the objective is to find input data that exercise paths with high similarity to the given reference path. Three functions for measuring similarity between paths are currently implemented in Algorithms 4, 5, and 6:

- length of the common path prefix or suffix,
- set distance of the paths' basic blocks, and
- multiset distance of the paths' basic blocks.

Using these basic functions the following additional measures for path similarity are defined. The *affix* similarity combines both *prefix* and *suffix similarity* into a single measure. The *combined similarity* combines the *affix*, *set*, and *multiset similarity* by assigning weights w_1, w_2, w_3 to each of them respectively. The calculation of distance between two execution path traces t_1 and t_2 by *combined similarity* is given in Equation 4.1.

$$\begin{aligned} sim_combined(t_1, t_2) = {} & w_1 \cdot sim_affix(t_1, t_2) \\ & + w_2 \cdot sim_set(t_1, t_2) \\ & + w_3 \cdot sim_multiset(t_1, t_2), \\ \text{with } & w_1 + w_2 + w_3 = 1, \\ \text{and } & w_1, w_2, w_3 \in [0, 1] \end{aligned} \tag{4.1}$$

A value of 1.0 similarity obtained by the *prefix*, *suffix*, *affix*, or *combined similarity* indicates that two paths are equal. In contrast, the measures for *set* and *multiset similarity* can give a similarity value of 1.0 even for two different paths if they traverse the same set, viz., multiset of basic blocks.

In the current implementation, experiments showed that paths with the highest degree of similarity, i.e., the highest fitness values, are obtained when using such settings for w_1, w_2, w_3 where the weight of the multiset similarity w_3 in Equation 4.1 is dominant. Furthermore, it was observed that the prefix and suffix based comparison is the most restrictive, while the multiset based comparison is less, and the set based comparison the least restrictive measure of similarity between two execution paths.

Algorithm 4 Measure of similarity between two path traces based on the length of their common prefix.

```
double
function prefix_similarity (linked_list_t *path1, linked_list_t *path2)
int i := 0;
linked_list_t *p1 := path1, *p2 := path2;
int max_len = MAX ( len(path1), len(path2));
int min_len = MIN ( len(path1), len(path2));
for i := 0; (p1 → val == p2 → val) AND (i < min_len); do
   i ++;
   p1 := p1 → next;
   p2 := p2 → next;
end for
return (i / (double) max_len);
```

Algorithm 5 Measure of similarity between two path traces based on the cardinality of intersection of their sets of unique basic blocks.

```
double
function set_similarity (linked_list_t *path1, linked_list_t *path2)
int match := 0, i := 0;
linked_list_t *p, *s1 := set (path1), *s2 := set (path2);
int max_len := MAX (len (s1), len (s2));
if len (s1) < len (s2) then
   exchange (s1, s2);
end if
p := s1;
for i := 0; i < max_len; i ++ do
   if p → val in s2 then
      match ++;
   end if
   p := p → next;
end for
return (match / (double) max_len);
```

Algorithm 6 Measure of similarity between two path traces based on the frequency of occurrence of each unique basic block in both paths.

```
double
function multiset_similarity (linked_list_t *path1, linked_list_t *path2)
i := 0, sum_min := 0, sum_max := 0;
int count1 := 0, count2 := 0;
linked_list_t *p, *s1 := set (path1), *s2 := set (path2);
int max_len := MAX (len (s1), len (s2));
if len (s1) < len (s2) then
   exchange (s1, s2);
end if
p := s1;
for i := 0; i < max_len;  do
   count1 := count (p, path1);
   count2 := count (p, path2);
   sum_min += MIN (count1, count2);
   sum_max += MAX (count1, count2);
   p := p → next;
   i++;
end for
return (sum_min / (double) sum_max);
```

4.3.2.4 Computation result

In the case of *computation result fitness*, the return value of the SUT main function is taken as the fitness value. It can be used for testing the optimization algorithm - in this case PSO - on well studied mathematical functions whose minima and maxima are analytically known. Such mathematical functions are an ideal environment for comparative benchmarks of different optimization techniques.

4.3.3 Other modules of the optimizer

Both the CPSO and DPSO algorithms utilize different topologies, metrics, and number-encoding schemes implemented as stand-alone modules of the *pso_wcet* framework. The functionality of each of the modules is described in Chapter 3. Exact implementations details for topology and encoding vary little from what is described in Subsections 3.4.1 and 3.4.5. The metric implementation, with the exception of few practical details, also adheres to the principles stated in Subsection 3.4.2.

4.3.3.1 Topology

The particle swarm optimizer contains the implementation of topologies described in Subsection 3.4.1; these are the *star*, *circle*, and *wheel topology*.

4.3.3.2 Encoding

The *binary* and *gray encoding* described in Subsection 3.4.5 are also implemented by the optimizer.

4.3.3.3 Metric

The optimizer implements metrics described in Subsection 3.4.2, the *Euclid*, *Manhattan*, and *absolute fitness distance based metric* in the following manner.

The CPSO algorithm applies the *Euclid* and *Manhattan metric* on particle positions in *Euclidean space* $\mathbb{X}^m$ (Subsection 3.2.1). The DPSO algorithm applies those same metrics on particle positions in *probability space* $\mathbb{P}^{m \times n}$ (Subsection 3.3.1).Variable m is equal to the size of the input data vector of the SUT main function. Variable n corresponds to the number of bits needed to encode each coordinate in Euclidean space. The *absolute fitness distance based metric* operates on the same datatype - the particle fitness value - in both algorithms. It provides the same distance measure for both the continuous and discrete binary PSO particles.

Additionally to metrics described in Subsection 3.4.2, distances between swarm particles can be defined as initial distances of their respective indices in the array data structure of the optimizer implementation. This is a static neighborhood relation, i.e., particles always have the same neighbors without regard to their actual distance in the search space. The computational overhead for this metric is low in comparison to other implemented metrics; there is no need to perform the $O(n^2)$ computation of distances between n particles in each iteration of PSO. This metric is called the *initial metric*, since the distances between particles are statically determined during particle initialization (Subsection 3.2.4).

4.4 Configuration

Configuration of the *pso_wcet* framework consists of specifying the parameters of the *particle swarm optimizer* and the *software-under-test*. Each SUT needs to have its own configuration entry, while the user can (but does not have to) change the default configuration of the optimizer. The default configuration together with the explanation of each parameter is given in Tables 4.1 and 4.2 for the optimizer and the SUT respectively.

4.5 Applications

4.5.1 Search for the longest execution path

One of the basic applications of *pso_wcet* is to generate input data that maximize SUT execution path length. In this context the SUT execution path is defined as a sequence of basic blocks, i.e. as a path of the CFG.

Although the longest execution path can be found by using graph-theory algorithms, e.g., a modified Dijkstra's algorithm as in [48], the practical value of using PSO for this task is in mapping the set of input data vectors to a certain execution path that they exercise. This provides an experimental guarantee of the execution path's feasibility.

4.5.2 Search for the WCET inducing execution path

Another application of *pso_wcet* is for estimating the WCET by generating input data that maximize the time cost of an SUT execution path. The difference between finding the most time consuming and the longest execution path is in the cost of basic blocks. Unlike execution path length, where each basic block has a cost of one, the exact *time cost* of each basic block depends on the number and

Parameter	Default Value	Section	Description
CIRCLE_K	2	3.4.1.2	Number of neighboring particles in circle topology
WHEEL_FOCAL_IDX	0	3.4.1.3	Index of the focal particle in wheel topology
INIT_V	0	3.2.4.3	Initial dimensional velocity of a CPSO particle
$\|V_MAX_CPSO\|$	X_{max}	3.2.8.1	Velocity limit of a CPSO particle
INIT_PROB	0.5	3.3.2.1	Initial bit probability of a DPSO particle
INIT_PROB_V	0	3.3.2.3	Initial bit probability velocity of a DPSO particle
V_MAX_DPSO	6.0	3.3.8	Argument to sigmoid function limiting the bit probability velocity of a DPSO particle
COGNITIVE_CPSO	2.0	3.2.8 3.2.10	Cognitive constant for CPSO
SOCIAL_CPSO	2.0	3.2.8 3.2.10	Social constant for CPSO
COGNITIVE_DPSO	1.0	3.3.6	Cognitive constant for DPSO
SOCIAL_DPSO	1.0	3.3.6	Social constant for DPSO
COL_DETECT_EN	0	3.5.2	Enables particle collision avoidance
ZERO_V_EN	1	3.2.9.1	Sets velocity vector of CPSO particle to zero if it leads outside of defined space
INERTIA_W	0.04	3.5.1.1	Inertia weight of particle

Table 4.1: Parameters of PSO Configuration

Parameter	Section	Description
SUT_MAIN	4.2.1	Main function of the software-under-test
SUT_ARGS_COUNT	4.2.1	Size of the input-data vector to the main function
SUT_ARGS_TYPE	4.2.1	Data type of the input-data vector
X_MIN	3.2.1 3.3.1	Minimum value of a single search space coordinate
X_MAX	3.2.1 3.3.1	Maximum value of a single search space coordinate
ALGORITHM	4.3.1	Determines the algorithm used for optimization
FITNESS_FUNCTION	4.3.2	Selects the fitness function
OPTIMIZATION_TYPE	4.3.2	Sets optimization to minimization or maximization
REFERENCE_PATH_INPUT	4.3.2.3	Input vector for inducing the reference execution path
DECIMAL_PLACES	3.4.5	Decimal precision of DPSO in Euclidean space

Table 4.2: Parameters of SUT configuration

type of assembly instructions it contains. Thus, each basic block can have a cost greater than one *unit of execution time*.

When using the particle swarm optimizer for WCET search it does not matter whether the worst-case execution time of instructions was obtained statically, from a processor model, or by measurements. If the WCET of instructions is to be obtained by measurements, the optimizer can be used to generate the input-data vectors. This application of *pso_wcet* is discussed in Subsections 4.5.3 and 4.5.4.

Assuming that the WCET costs of basic blocks are provided, it is possible to maximize SUT execution time by the particle swarm optimizer. The use of predefined values for the time costs of processor (or in the current implementation virtual machine) instructions, may provide a safe WCET estimate on simple processor architectures. The necessary condition for *safety* of the WCET estimate derived in this way is a processor system without timing anomalies.

Since PSO is a heuristic optimization method and not an exact algorithm, it cannot be guaranteed that the path found by it is indeed the most time consuming execution path of the SUT. The benefit of PSO is that the found path is guaranteed to be feasible; the proof of paths's feasibility is offered by the input data exercising it.

It can be concluded that the PSO algorithm applied to the problem of WCET search can only provide a lower estimate of WCET. However, the feasibility of the execution path generating that estimate is guaranteed. In contrast, methods for static WCET calculation: *explicit path enumeration*, IPET, and *tree-based calculation* can provide a safe WCET estimate, assuming that the necessary condition of hardware free from timing anomalies is met. The costliest path obtained with static methods can be unfeasible at run-time which makes overestimation of WCET possible.

4.5.3 Exercising a specific execution path

The user of *pso_wcet* can specify a certain *reference path* and use a *path similarity fitness function* to maximize similarity between the reference path and paths induced by the optimizer-generated input data. Currently implemented measures of path similarity are discussed in Subsection 4.3.2.3.

Input data vectors obtained by maximizing the similarity of execution paths to that of the reference path can be used to perform execution time measurements of the reference path. Different input vectors are needed in order to observe the variation, i.e. jitter, of execution times for processor instructions with non-constant timing properties. In this case the observed variation of execution times is the result of data-dependent execution time of certain instructions. Additional variation in instruction execution time, on more complex processor architectures, can be a result of *execution history* [4], but this phenomenon is not considered here.

4.5.4 Exercising an execution path neighborhood

Similarly to finding input data for a reference execution path, it is also possible to generate data that exercise paths in the reference paths's neighborhood. The *path neighborhood* is defined as a set of execution paths that satisfy a certain similarity criterion with regards to the reference path. The *similarity criterion* is usually specified by the user; it can be an interval of the result of the path similarity fitness function. An exemplary fitness interval $0.9 \leq f < 1$ would consider all paths with a dissimilarity of up to 10% to be inside the reference path's neighborhood.

When the *vectorout* option of *pso_wcet* is invoked, the position of all particles that satisfy the fitness criterion is saved at the end of every iteration. The result is a set of data vectors that exercise paths in the defined neighborhood.

Furthermore, were the concept of path neighborhood extended with the definition of a *program segment*, e.g., using a segmentation scheme proposed in [49], it would be possible to obtain input data exercising a segment in the same way as input data exercising a neighborhood are generated in the current implementation. The obtained input data could be used for extensive measurements of segments, i.e., measurement-based derivation of their worst-case execution time. Incorporating a segment definition in *pso_wcet* is a subject of further work.

4.5.5 Checking execution path feasibility

If some SUT execution path is structurally or semantically not feasible, then it is automatically disqualified from being a WCET inducing execution path.

The problem of execution path feasibility, defined in Subsection 2.3.5, can be solved by *pso_wcet* to the following extent. The user provides a trace of the path that needs feasibility checking inside a text file. The framework reads this execution path and uses it as the reference path. Combined with one of the path similarity fitness functions[5] from Subsection 4.3.2.3, it searches for the data vector that exercises the reference path by maximizing path similarity.

If at the end of the run, a data vector whose execution path has a similarity of 1.0 with the reference path is found, then the user specified reference path is deemed feasible. If that is not the case, no statement about the path's feasibility is made.

[5] All implemented fitness functions for path similarity can be used for checking execution path feasibility except the set and multiset similarity. These two functions may not be used, since they can give a similarity of 1.0 even for non-equal paths.

4.6 Related Work in Software Testing

Software testing implies dynamic execution of a software component with a set of input data vectors under specified execution conditions. Results of execution are recorded and evaluated against the software component's *specification.*

4.6.1 Types of software testing

The specification of a software component in real-time systems contains *functional* as well as *temporal requirements.* Thus, it can be spoken of the functional and the temporal part of the component's specification.

4.6.1.1 Testing for functional requirements

Adherence to the functional specification is usually verified by using a set of *test cases.* A test case in real-time systems contains an input data vector, a description of the execution conditions, and a set of expected results [50, p.251]. It embodies the relationship between the component's *input*, *output*, and *change of state.* Failing to satisfy a correctly implemented test case shows that the implementation of the software component differs from its specified functionality.

All test cases are important in functional testing; software components need to be regularly tested with them during the development cycle in what is known as *regression testing.* Failing to satisfy a single test case means that the component does not function according to its specification. The absence of failed test cases, however, does not prove the absence of functional errors. It merely increases the certainty of their absence.

4.6.1.2 Testing for temporal requirements

In the case of *temporal testing* the goal is to find the test case, i.e., the input data vector which induces the WCET of the software component. Extensive coverage of input data as well as the execution paths inside the SUT is necessary for any reasonable certainty in the lack of optimism of the WCET estimate.

If the WCET estimate obtained by testing is higher than the one specified in the temporal specification, then the SUT fails to satisfy it. If, however, the WCET estimate is lower than what is specified, this still does not prove that the software component complies with its temporal requirements. A proof of that could only be given by static WCET analysis. Still, a WCET estimate obtained by testing can be valuable to software developers, as it can point to problems with temporal behavior early on in the development cycle [30].

4.6.2 Algorithms for generating input data

The input data used for functional and temporal testing can be generated automatically by suitable heuristic algorithms. Test data generated with a genetic algorithm have been used in [30] for numerous kinds of functional testing (black box, white box, gray box) and limited end-to-end measurements of WCET. In [29], a genetic algorithm has been applied solely for testing the temporal properties (WCET, BCET) of a real-time system component. In the original work about white box functional testing [51], a gradient descent method for function minimization has been applied to generate test data. In a more recent work [52], both PSO and a genetic algorithm have been implemented to generate test data for gray-box functional testing. The authors have compared the two algorithms and have concluded that PSO outperforms genetic algorithms in most of the cases.

4.6.3 Types of fitness functions

Apart from the different algorithms used to generate input data, testing methods also differ by the applied fitness functions. The traditional division of functional testing is that into *black box*, *white box*, and *gray box testing*. Among these methods, white box and gray box concepts have the most relevance for testing of temporal behavior.

4.6.3.1 Black box testing

Black box testing checks whether certain data vectors provided as input cause the software component under test to give correct data as output. The test cases used in this kind of testing have to capture the relevant *input-output relationships* from the functional specification of the software component.

The black box approach has little relevance for estimating WCET; it deals only with the functional requirements of the SUT.

4.6.3.2 White box testing

White box testing deals with *internal operation* of the software component. It checks whether the provided input data exercise the correct execution path. A prerequisite for this kind of testing is knowledge about the internal structure of the software component, hence the name 'white' box testing. Information about the internal structure can be captured by the CFG. White box testing is used mostly to detect errors in the implementations of program logic, e.g., coding errors which invalidate the correct flow-of-control.

The main goal of white box testing is to achieve high *coverage of code* with tests. Code coverage can be measured in different ways, most simplistically, by

the number of exercised statements. When single statements are grouped into basic blocks, by using the definition stated in Subsection 2.3.1, code coverage can be measured in terms of the number of exercised basic blocks. The coverage of basic blocks in turn leads to the coverage of execution paths. Each execution path explicitly states a sequence of executed basic blocks. The evaluated branch conditions can be implicitly inferred from the execution path. However, in some applications, notably in [29], it can be advantageous to measure code coverage in terms of branch coverage, i.e., by the number of exercised branch predicates.

Since white box testing aims at high code coverage, the generated input data can be reused for temporal testing. Usually, only a small subset of obtained execution paths will cause the software component to exhibit its worst case temporal behavior. High code coverage provides a degree of certainty that the WCET estimate obtained by testing is not unreasonably low.

4.6.3.3 Gray box testing

In contrast to white box testing, the task of gray box testing is not in achieving high code coverage of the entire software component. Its task is to exercise the software component with such input data that cause certain *conditions of interest* to occur during execution. For example, an exception condition may need to be caused in order to test whether the exception handling mechanism is correctly implemented.

Often, WCET can be induced by input data that trigger an exception [9, p. 8]. Because of the comparatively rare occurrence of such input data, these need to be generated selectively and separately from input data used in white box testing. The fitness functions used to generate test data in gray box testing are based on *goals*, i.e., on targeting specific parts of program code for execution. Goals can be single statements, branch predicates, or execution paths.

The framework for software testing presented in [30] implements an elaborate goal mechanism for gray box testing of exceptions. A similar mechanism is implemented in *pso_wcet* in the form of searching for some specific execution path; this application was discussed in Subsection 4.5.3.

4.7 Related Work in Measurement Based Timing Analysis

A summary of *pso_wcet* and two related methods for measurement-based timing analysis (MBTA) is given in Table 4.3.

Both the measurement based method presented in Atanassov [27] and the current implementation of *pso_wcet* belong to the class of end-to-end measurements.

	Atanassov, [27]	*pso_wcet*	Wenzel, [33]
Method type	end-to-end	end-to-end	hybrid
Source code analysis	-	basic blocks, CFG	parse tree, basic blocks, CFG, segmentation
Input data generation	GA	CPSO, DPSO	GA, model checking
Derivation of execution times	measurement of single instructions	synthetic cost of single instructions	measurement of instruction sequences
Basic code unit in calculation	whole program	basic block	segment
WCET Calculation technique	heuristic longest path search	heuristic longest path search	graph based longest path search and IPET

Table 4.3: Comparison of related work in MBTA.

In contrast, the approach in Wenzel [33] uses an intermediate stage before calculating the global WCET, namely the local WCETs of program segments. The calculation step is carried out using IPET or with a graph based algorithm for longest-path search. Thus, the approach in [33] can be classified as a hybrid method.

In the case of *pso_wcet* and [27], the search for the longest execution path is performed by a heuristic search technique. The search proceeds on a best effort basis and yields a lower WCET bound.

Both *pso_wcet* and [33] use local WCET bounds of basic code units in the calculation step. This can lead to overestimation of global WCET, since the execution times of code units can vary over a wide spectrum, depending on the current execution context [53]. In [27], the WCET estimate is obtained by executing the whole program on a specific hardware platform. In contrast, the approach of *pso_wcet* is not platform specific; it uses synthetic WCET bounds of single assembly instructions. The synthetic bounds can in principle be replaced with bounds obtained by measurements or from a timing model. Furthermore, a reduction in computational complexity of *pso_wcet* could be achieved by applying one of the program segmentation technique used in [33] or [49]. This would enable more efficient analysis of larger programs.

4.8 User Manual

This section contains the print out of *pso_wcet* manual pages.

NAME

pso_wcet – a framework for worst-case execution time and execution path analysis of C programs

SYNOPSIS

pso_wcet –p *<particles>* [**–i** *<iterations>* |**–-tf** *<target-fitness>*] [**--fitness** *pathlen* |*pathcost* |*pathaffixcmp* |*pathsetcmp* |*pathmultisetcmp* |*pathcombinedcmp* |*fresult*] [**--refpath** *<file>*] [**--algorithm** *cpso* |*dpso* |*random*] [**--optimization** *min* |*max*] [**--topology** *star* |*circle* |*wheel*] [**--metric** *init* |*euclid* |*taxi* |*fitness*] [**--encoding** *gray* |*binary*] [**--cognitive** *<value>*] [**--social** *<value>*] [**--vmax** *<value>*] [**--vectorout** *[<count>] [--fg |--fl] <fitness>*] [**--of** *<output>*] [**–s** | **–b** | **–h**]

OPTIONS

Optimizer options

–p *particles*

Use this options to set the number of particles in search space.

–i *iterations*

Uses the maximum number of simulation iterations as the termination condition.

–-tf *target-fitness*

The simulation terminates upon reaching a certain fitness.

–-fitness

pathlen Uses execution path length in basic blocks as the fitness function.

pathcost This fitness function sums the time costs of basic blocks in the execution path.

pathaffixcmp This fitness function compares the prefix and suffix of each execution path with that of the reference path.

pathsetcmp This fitness function compares the basic block set of each execution path with that of the reference path.

pathmultisetcmp Compares the multiset of basic blocks of each execution path with that of the reference path.

pathcombinedcmp Uses a combination of *pathaffixcmp* , *pathsetcmp* , and *pathmultisetcmp* for the fitness function.

fresult The return value of the SUT main function is used as the fitness function.

–-optimization

min The fitness value is minimized.

max The fitness value is maximized.

–-algorithm

cpso Optimizer uses the continuous PSO algorithm.

dpso Optimizer uses the discrete PSO algorithm.

random Optimizer uses random search.

I/O options

–-refpath *file*

An execution path from a user specified file is used as the reference path. If this option is not used, the reference path is obtained at runtime using the input data from *subject/subject.h* .

–-vectorout

count Specifies the number of input data vectors to be produced as output.

−-fg fitness Produces data vectors whose fitness is greater than the specified value.

−-fl fitness Produces data vectors whose fitness is lesser than the specified value.

−-of *file*
Specifies the output destination of all simulation results.

PSO options

−-topology

star The star topology in PSO models democratic decision making; all particles form one big neighborhood.

circle The circle topology in PSO defines two nearest particles in the search space as neighbors of some particle.

wheel The wheel topology in PSO models a hierarchical organization using a single focal particle. Other particles on the periphery are in the neighborhood relation only with the focal particle.

−-metric

init The initial metric uses the position of particles in memory for determining their distance.

euclid The Euclidean metric determines the distance of particles by measuring their distance in search space with the Euclid's formula.

taxi The Taxi or Manhattan metric determines the distance between particles in search space with the Manhattan formula.

fitness This metric determines the particle distance using the absolute difference of their respective fitness values.

−-encoding

gray This option encodes bits using Gray encoding in DPSO and random search.

binary This option encodes bits using standard binary encoding in DPSO and random search.

−-cognitive *value*
Sets the double value of the cognitive constant. Recommended settings are inside the interval [0,3].

−-social *value*
Sets the double value of the social constant. Recommended settings are inside the interval]0,3].

−-vmax *value*
Sets the velocity limit for CPSO and DPSO; recommended are double values from the interval [1,4] and [3,10] respectively.

Modes of operation

–s Silent mode, optimizer displays the end result of simulation only.

–b Benchmark mode, optimizer displays the best particle of each iteration.

–h Help mode, displays this help.

DESCRIPTION

pso_wcet is a framework for generating input data to exercise the flow-of-control of C programs. The input data are generated with *Particle Swarm Optimization* using different fitness functions. Upon execution, each input data vector exercises a certain program path consisting of a sequence of basic blocks.

Applications

Based on the applied fitness function the applications of the framework include the following:

1. Maximizing the longest execution path in terms of the number of basic blocks
2. Maximizing the longest execution path in terms of the execution time cost (WCET)
3. Generating test data that exercise a specific execution path
4. Exercising execution paths in the neighborhood of some specified path
5. Empirically checking the feasibility of a specified path

SUT Integration

Preparing a new software component for testing by the framework involves taking the following steps:

1. Write the software under test (SUT) in a file location **<sut_filename>**, e.g. in *subject/other/test1.c* . The main function of the SUT has to conform to the signature:

double<*func_name*>(int**count,*[double|int]*array[])* .

2. Write the runtime configuration of the SUT in *subject/subject.h* using the following template:

```
#ifdef <sut_id>
  #define SUT_MAIN_FUNCTION <func_name>
  #define SUT_ARGS_COUNT <int>
  #define SUT_ARGS_TYPE <int>
  #define DECIMAL_PLACE <int>
  #define X_MIN <double>
  #define X_MAX <double>
  #define ALGORITHM <int>
  #define FITNESS_FUNCTION <int>
  #define OPTIMIZATION_TYPE <int>
  #define REFERENCE_PATH_ARGS <arg type array>
#endif
```

Also write the declaration of the SUT main function into *subject/subject.h* . The name of this function has to be different from **<sut_id>** .

3. Include the following entry into *subject/subject.c :*

```
#ifdef <sut_id>
  #include "<sut_filename>"
#endif
```

4. Define the current SUT in *subject/subject.h* with:

```
/* current SUT main function */
#define <sut_id>
```

5. Recompile the project with: **make -f Makefile.man purge all**

If there have been any errors in the compilation arising from wrong configuration in the previous steps correct them. Sometimes clang cannot handle more complex C constructs used in the SUT, e.g. nested expressions in boolean evaluation. The solution to this is to simplify the syntax of such expressions, i.e. to subdivide them in a few simpler expressions.

EXAMPLES

./pso_wcet -p 30 -i 300 --fitness pathlen

Maximizes the execution path length of the current SUT in terms of basic blocks. The simulation uses 30

PSO particles and lasts for 300 iterations.

./pso_wcet -p 30 --tf 12321

Maximizes the execution time until the target fitness of 12321 time units is reached.

./pso_wcet -p 30 -i 300 --fitness pathmultisetcmp --vectorout --fg 1.0 -s

Finds additional input data that exercise the reference execution path. The input data inducing the reference execution path are defined in *subject/subject.h* .

./pso_wcet -p 30 -i 300 --fitness pathmultisetcmp --refpath results/optimum-path-bubblesort.dat --vectorout --fg 0.99 --fl 1.0 -s

Finds input data vectors that induce execution paths in the neighborhood of the user specified path.

./pso_wcet -p 30 -i 300 --optimization min --fitness pathmultisetcmp --refpath results/optimum-path-bubblesort.dat --vectorout --fl 0.9 -s

Finds input data vectors that induce paths with least similarity to the user specified path.

/pso_wcet -p 30 -i 300 --fitness pathmultisetcmp --refpath results/unfeasible-path-bubblesort.dat --vectorout --fg 0.999 --fl 1.0 -s

Checks whether a certain path is feasible. If no input data vectors with fitness 1.0 are found, the path is empirically deemed as unfeasible.

./pso_wcet -p 30 -i 300 --algorithm dpso --topology circle --metric fitness --encoding binary --cognitive 1.1 --social 2.9 --vmax 4.0

Maximizes SUT execution time with the discrete PSO algorithm using non-default settings for the topology, metric, encoding, cognitive and social constant, and maximum velocity.

FILES

subject/subject.c
File includes C source code of the software-under-test.

subject/subject.h
File contains different optimization configurations.

subject/subject.bcsi
File contains instrumented LLVM code of the software-under-test; it also contains basic block IDs.

subject/bbcost.dat
File contains the execution time of basic blocks.

AUTHOR

Miljenko Jakovljevic (miljenko.jakovljevic@gmail.com)

4.9 Summary

The *pso_wcet* framework consists of code instrumentation, tracing of execution, and the PSO optimizer. A SUT is written in the C programming language, and its instrumentation is performed on the intermediate code level in LLVM virtual assembly. An execution trace is obtained by running the SUT binary, compiled from instrumented code in LLVM, with the input data vectors provided by the PSO optimizer.

The main function of the SUT has to conform to the signature specified by *pso_wcet*. Furthermore, the framework needs to be configured with SUT specific parameters before generation of input data can take place. Default values can be used for parameters that are relevant only for the PSO simulation, although these may not yield optimal results in all instances.

Applications of *pso_wcet* include finding the longest and worst-case execution path, generating data vectors that exercise a certain reference path, as well as exercising neighborhoods of the reference path, and checking execution path feasibility.

The generated data vectors can be used for time measurements on real hardware. Different input data are needed to observe the data dependent variance of execution time for some path. Given sufficient computational resources, the *pso_wcet* framework can provide such input data.

Chapter 5

Experiments

The plan for experimental evaluation of *pso_wcet* was the following: in the first phase, the three algorithms DPSO, CPSO, and *random search* were used to optimize a given software-under-test. The optimization was carried out with different settings for *topology*, *metric*, and *encoding*. Other PSO parameters were set to default values, shown in Table 5.1.

In the second phase, the value of each PSO parameter: c_1, c_2, V_{max}, *particle count*, and *iteration count* was changed while using the default values for other parameters. Topology, metric, and encoding were set to values which had optimal performance in the previous phase. The task of the second phase was to find improvements of default PSO parameters.

In the third phase, the experience in parameterization obtained from the first two phases was applied to estimate the WCET, i.e., to maximize the execution time of the SUT. Different values for V_{max} were tested as well.

Each software-under-test can be a unique problem. Hence, follows that a certain PSO configuration does not have to be optimal for all SUTs. The goal of the above three-phase approach was to finetune the PSO configuration for specific SUTs and, possibly, to draw more general conclusions about the right parameterization of PSO.

Parameter	CPSO	DPSO
Cognitive constant c_1	2.0	1.0
Social constant c_2	2.0	1.0
Maximum velocity limit V_{max}	16.0	6.0
Zero velocity outside search space	True	-

Table 5.1: Default configuration of CPSO and DPSO.

Software under test	`bubble sort`
Fitness function	execution path cost - WCET
Optimization type	maximization
Type of input vector	[`int`]
Size of input vector	20
Search space	$[-16, 15]^{20}$

Table 5.2: Configuration of the optimization problem `bubble sort`.

PSO particles	30
Iterations per run	100
Total runs	100

Table 5.3: General configuration of PSO for experiments on `bubble sort`.

5.1 Maximization of the WCET Estimate

The first software under test was the `bubble sort` algorithm with source code from Listing 4.1. Its configuration, assembled in Table 5.2, was the same for all experiments. Furthermore, the general configuration of PSO that was used in Experiments 1 and 2 on bubble sort is summarized in Table 5.3. The selected *particle* and *iteration count* was observed to bring good fitness results, sometimes even the optimum, while using modest computational demands on the testing platform. The selected number of *experiment runs* was deemed sufficient, on the whole, to negate the effects of randomness.

5.1.1 Experiment 1: Effect of topology, metric, and encoding

The first experiment on `bubble sort` dealt with the effect of topology, metric, and encoding on the performance of PSO based WCET estimation. This effect was investigated independently of other PSO parameters. Experiment settings are shown in Table 5.3, and the results are assembled in Table 5.4.

Both the CPSO and DPSO algorithm outperformed *random search*[1] in all the experiment runs. The *star* and *wheel topology* induced the two highest WCET values as well as the two highest mean WCET values, for both CPSO and DPSO. Furthermore, *gray encoding* induced comparable or higher maximum and mean

[1] *Random search* was implemented as a special case of DPSO where bit probabilities always have a value of 0.5.

WCET than *binary encoding*, for all combinations of topology-metric with DPSO and random search.

For the CPSO algorithm, the star topology induced the highest mean WCET and the smallest standard deviation. The wheel topology induced the maximal WCET value.

For the DPSO algorithm, the star topology, initial metric, and gray encoding induced the highest maximal and the highest mean WCET value with the smallest standard deviation. Overall, the best WCET estimate of DPSO was higher than that of CPSO.

Based on these results, the star topology, initial metric, and gray encoding were selected for further use in Experiment 2.

Experiment 1: `bubble sort`					
WCET estimate by *Random search*					
Encoding	Minimum	Maximum	Mean	Stdev	Mean rank
binary	**11187**	**11607**	11351.64	*91.84*	*2*
gray	**11187**	11586	**11362.35**	**82.09**	**1**

WCET estimate by CPSO						
Topology	Metric	Minimum	Maximum	Mean	Stdev	Mean rank
star	initial	**11880**	*12069*	**11970.3**	**38.52**	**1**
wheel	initial	*11775*	**12090**	*11903.94*	52.81	*2*
circle	initial	11754	12048	11856.48	54.41	6
circle	fitness	11733	12027	11862.78	54.42	5
circle	euclidean	*11775*	12048	11881.05	54.91	3
circle	manhattan	*11775*	*12069*	11872.65	*52.5*	4

WCET estimate by DPSO							
Topology	Metric	Encoding	Minimum	Maximum	Mean	Stdev	Mean rank
star	initial	binary	*12069*	**12300**	*12206.13*	40.1	*1b*
star	initial	gray	**12132**	**12300**	**12257.58**	**28.47**	**1a**
wheel	initial	binary	11943	*12237*	12097.77	51.06	2b
wheel	initial	gray	12027	*12237*	12136.41	45.69	2a
circle	initial	binary	11943	12153	12053.04	41.48	5b
circle	initial	gray	11985	12174	12073.2	38.34	3a
circle	fitness	binary	11838	12090	11968.2	44.17	6b
circle	fitness	gray	11922	12132	11998.44	47.24	6a
circle	euclidean	binary	11964	12153	12064.59	*37.55*	5a
circle	euclidean	gray	11964	12195	12068.58	40.49	3b
circle	manhattan	binary	11964	12174	12068.16	44.96	4a
circle	manhattan	gray	11943	12174	12066.48	46.56	4b

Table 5.4: WCET estimate of `bubble sort` by DPSO, CPSO, and *random search*. Different topology/metric/encoding combinations were evaluated. The entries in bold are the best, and the entries in italics are the second best for each column.

Algorithm	Topology	Metric	Encoding
CPSO	star	initial	-
DPSO	star	initial	gray

Table 5.5: Special configuration of PSO for Experiment 2 on `bubble sort`. The combination of topology, metric, and encoding with optimum performance in Experiment 1 was selected for further application in this experiment.

5.1.2 Experiment 2: Effect of other PSO parameters

The second set of experiments investigated the following PSO parameters for their effect on the WCET estimate: *cognitive constant* c_1, *social constant* c_2, *maximum velocity* V_{max}, *particle count*, and *iteration count*. The configuration of Experiment 2 is displayed in Tables 5.3 and 5.5.

In each experiment instance, the PSO parameters that were not currently being surveyed were set to default values from Table 5.1. The use of these values for c_1, c_2, and V_{max} is reported in PSO literature, e.g., in [35, 39, 41, 37] for CPSO, and in [38, 43] for DPSO.

5.1.2.1 Cognitive and social constant

Figures 5.1 and 5.2 show the effect of cognitive constant c_1 and social constant c_2 on the performance of CPSO and DPSO in estimating WCET. The x and y axes display different values for c_1 and c_2 while the fitness values - WCET - are depicted in a topographic manner. From these figures the following observations can be made.

For the CPSO algorithm, the highest mean fitness was achieved on the narrow strip in the lower left corner of the diagram. This corresponds to the setting of $c_2 = 1$ and $c_1 \in [1, 1.5]$. The fitness from this c_1-c_2 area had comparatively high standard deviation. The lowest mean fitness was exhibited by the settings $c_2 \in [1.5, 3]$ and $c_1 \in [1, 1.5]$. It is depicted as the blue area in the upper left corner of the diagram. The default learning factors $(c_1, c_2) = (2.0, 2.0)$ induced mediocre mean performance and low standard deviation.

In contrast to CPSO, the DPSO algorithm had a large area with optimal c_1, c_2 values. Roughly, the parameters which induced the highest mean fitness were $c_1 > 0.5$ and $c_2 > 1$. Values of $c_2 < 0.4$ induced the lowest mean fitness. The default setting of $(c_1, c_2) = (1, 1)$ induced good mean fitness and mediocre standard deviation. The standard deviation could be reduced, while retaining the optimal mean fitness, by moving the parameters (c_1, c_2) in the direction $(1, 1) \rightarrow (2, 2)$.

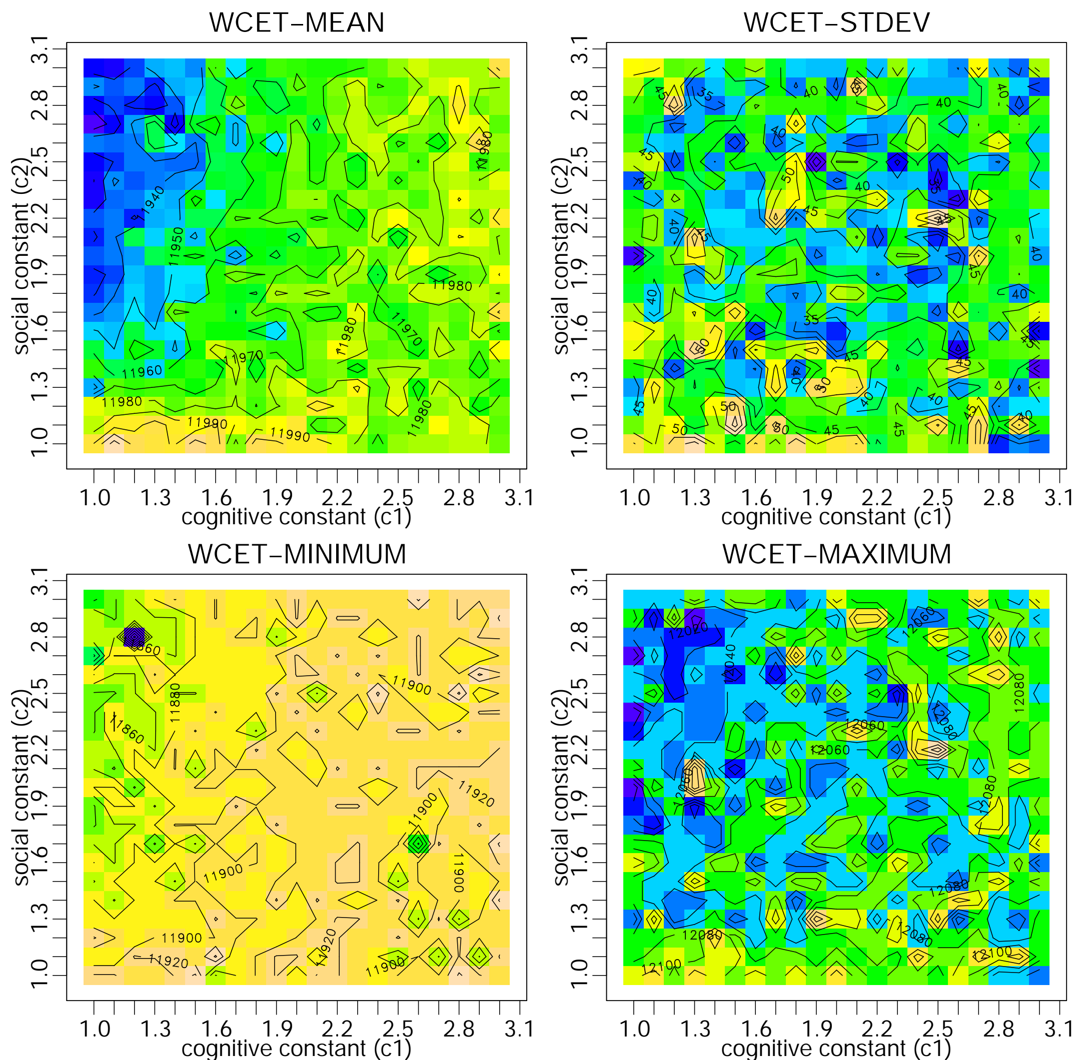

Figure 5.1: Influence of the cognitive and social constant on the WCET estimate of *bubble sort* that was obtained by CPSO.

5.1.2.2 Maximum velocity

The effect of maximum velocity, V_{max}, on the WCET estimates obtained by CPSO and DPSO is shown in Figures 5.3 and 5.4 respectively.

To recall, V_{max} controls the relationship between global and local search of PSO. High maximum velocity allows more global search at the beginning and less local search towards the end of the algorithm run. In contrast, lower maximum velocity allows particles to cover a smaller area of search space at the beginning,

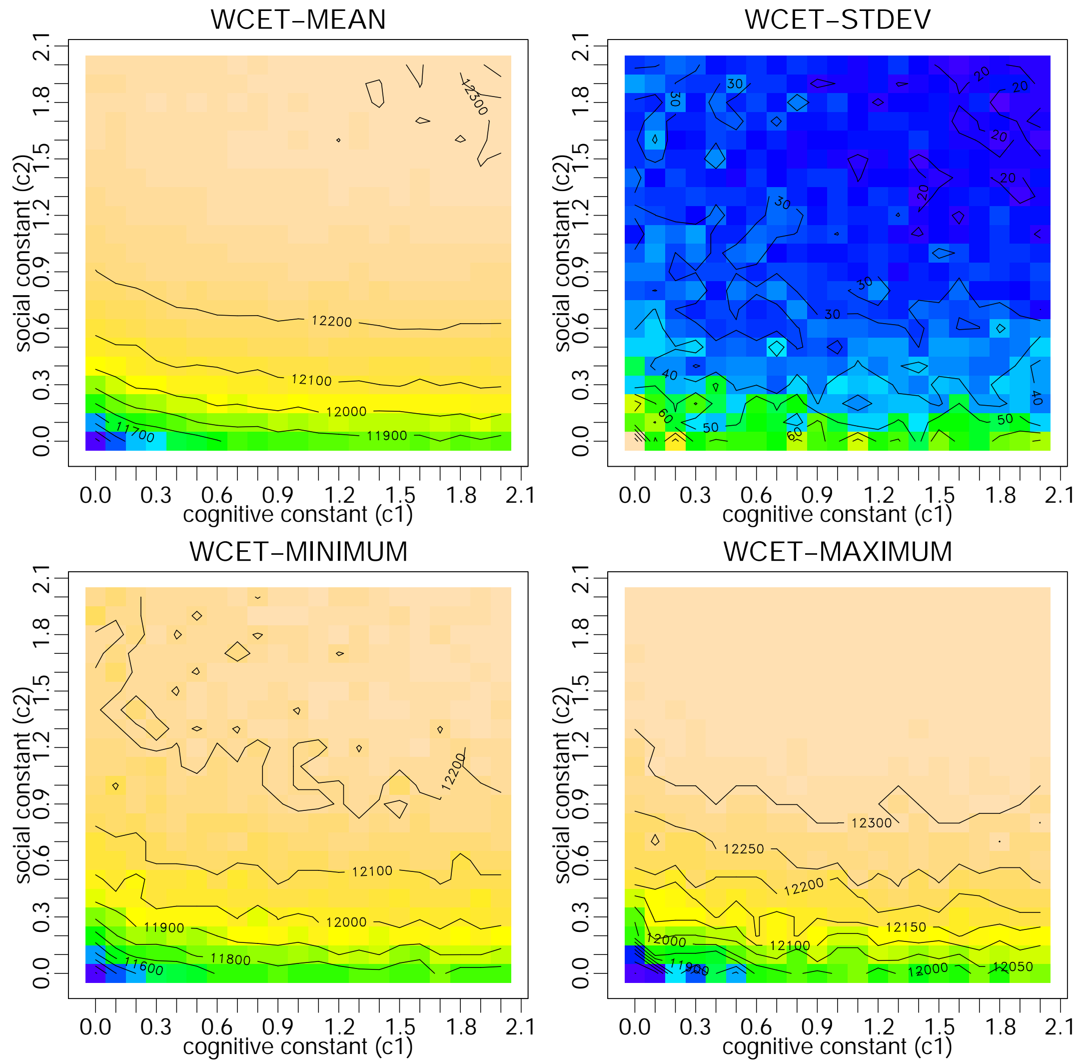

Figure 5.2: Influence of the cognitive and social constant on the WCET estimate of `bubble sort` that was obtained by DPSO.

but it allows more local exploration towards the end of optimization.

For the CPSO algorithm in Figure 5.3, $V_{max} = 2$ induced the maximum mean fitness and highest standard deviation. The default value, $V_{max} = X_{max} = 15$, provided mediocre mean fitness and medium standard deviation. It would seem prudent not to use values of $V_{max} > 5$, since they induced reduced mean fitness. Furthermore, values of $V_{max} \in [2, 5]$ would have the benefit of allowing more local search in the finishing iterations of CPSO.

In contrast to CPSO whose mean fitness decreased after $V_{max} > 2$, the mean

fitness of DPSO increased after $V_{max} > 0$, as shown in Figure 5.4. For values of $V_{max} \in [4.0, 10]$, the mean fitness remained static. The standard deviation induced for this interval was lower than the corresponding deviation for $V_{max} < 4$. This experiment would indicate that $V_{max} \in [4.0, 10]$ causes DPSO to perform a more systematic search than for $V_{max} < 4$, since these values induced the highest fitness and lowest standard deviation. The minimum value from the proposed interval, i.e. $V_{max} = 4$, could be optimal when optimization is performed with more iterations than in this experiment, since it would allow most local search.

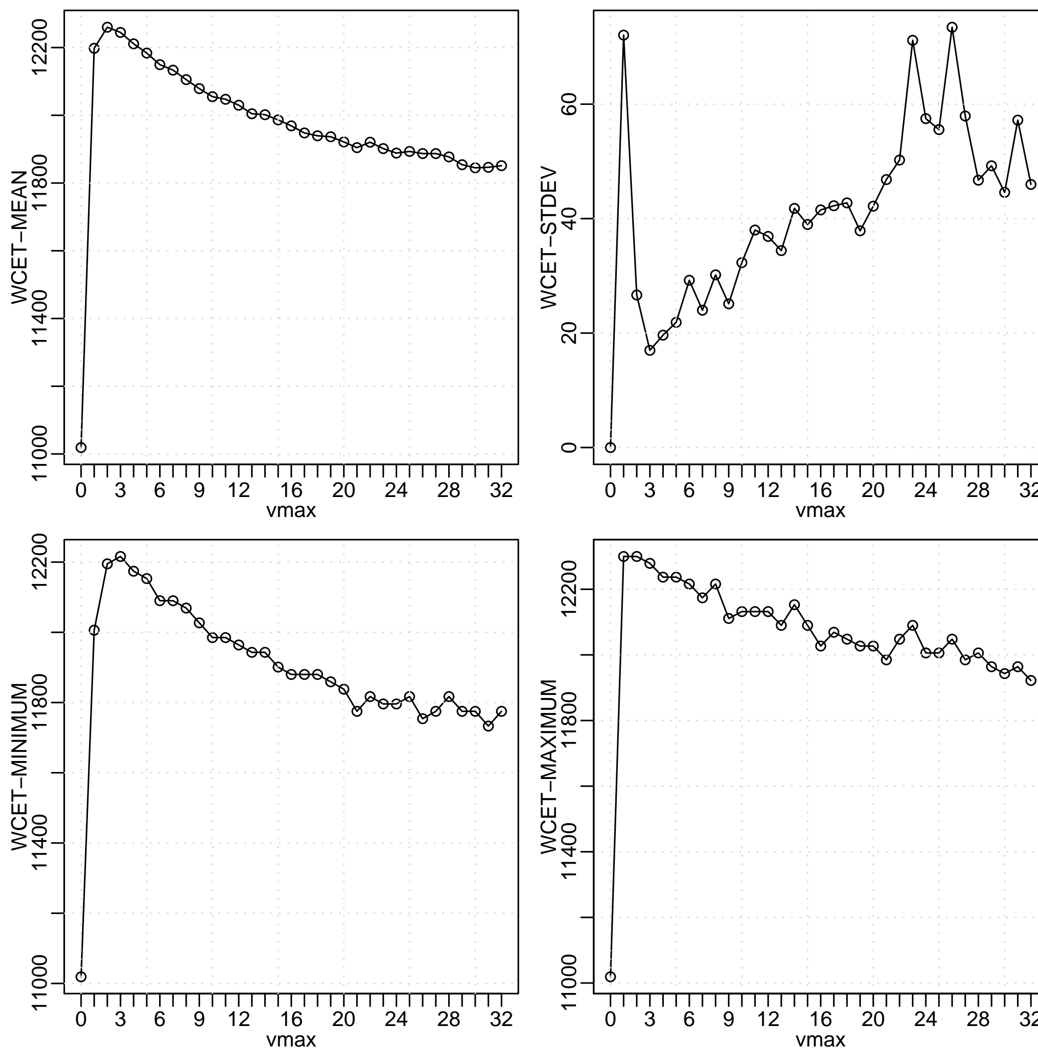

Figure 5.3: Influence of maximum velocity, V_{max}, on the WCET estimate of `bubble sort` obtained by CPSO.

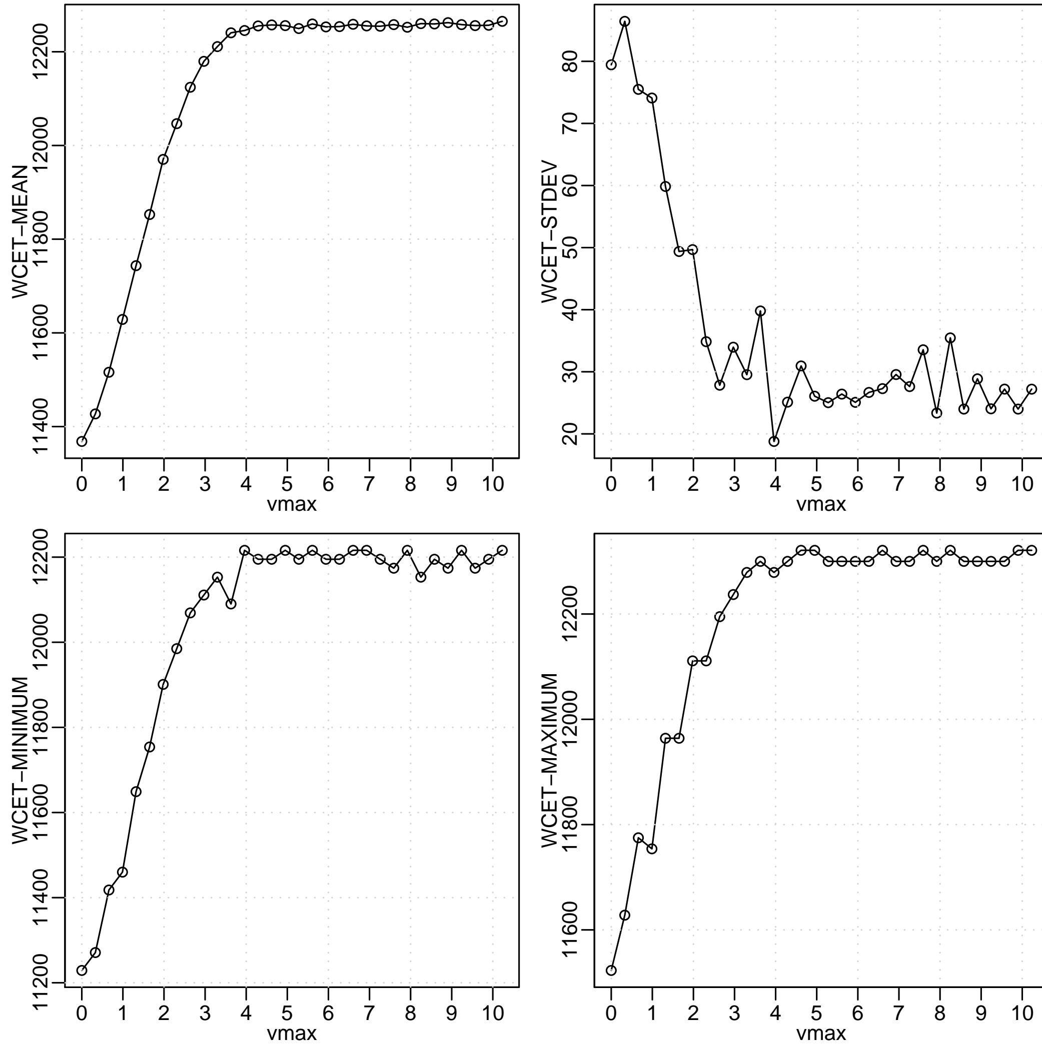

Figure 5.4: Influence of maximum velocity, V_{max}, on the WCET estimate of `bubble sort` obtained by DPSO.

5.1.2.3 Particle count

The effect of *particle count* on the WCET estimate of CPSO and DPSO is shown in Figures 5.5 and 5.6 respectively.

It should be noted here that a larger particle count can achieve higher search space coverage at a single instant. However, this comes at the price of increased computational effort, since each particle needs to be updated independently by the optimizer. Furthermore, the computational effort of coordinating the swarm,

i.e. calculating the *gbest* position and the *neighborhood relation*, increases with particle count. The same computational effort can be better spent by increasing the number of iterations than by increasing the number of particles after a certain *particle utility limit*. This limit can be defined as a point after which the linear increase of *particle count* results in less than a linear increase of fitness for some *iteration count*. In the current implementation, *particle count* n increases the overhead of one PSO iteration by at least a factor of $\Omega(n^2)$.

The CPSO algorithm in Figure 5.5 offered little improvement of mean fitness for *particle count* > 30; standard deviation remained bounded between 30 and 50 in this interval. Thus, there is little reason to justify the computational effort necessary for more particles.

In contrast to CPSO, the DPSO algorithm reached the *particle utility limit* somewhat later, at 50 particles. After this point there was not enough improvement of fitness to justify the computational expense; the linear increase of particles above this number did not result in the likewise proportional increase of fitness.

For DPSO, a better improvement could usually be achieved by a higher *iteration count*. By comparing the experiments shown in Figures 5.6 and 5.8 it can be observed that, increasing the *particle count* improved the fitness of DPSO less than the corresponding increase of *iteration count*.

It can also be observed from Figure 5.6 that the standard deviation continually dropped with increased particle count in the case of DPSO. This would indicate that the diversity of the swarm decreases proportionally with the size of the swarm, i.e., the convergence of particles on the same coordinates increases. In contrast, the standard deviation of CPSO in Figure 5.5 shows that the diversity of the swarm remained bounded for *particle count* > 10.

5.1.2.4 Iteration count

The effect of *iteration count* on WCET estimates obtained by CPSO and DPSO is visible in Figures 5.7 and 5.8 respectively.

Intuitively, when the algorithm is run with a higher iteration count it tends to obtain higher WCET estimates. However, after a certain number of iterations little or no improvement of fitness may occur. It is hypothesized that this condition is caused by particles converging on some region of search space with local optima. Another possibility may be that there are simply too few coordinates in search space that induce the optimum fitness - WCET, i.e. that the optimal coordinates are too sparse. If all particles converge on a local optimum it can be effective to restart the algorithm. Particles can then perform search from different starting conditions, which can lead to convergence on those regions of space with global optima.

The CPSO algorithm in Figure 5.7 achieved high fitness gains for *iteration*

count $\leq$ 30. For *iteration count* $>$ 30, improvements occurred at a much lower rate. Standard deviation remained within constant bounds in this interval.

The progressive improvement of mean fitness by DPSO in Figure 5.8 was much steadier than that of CPSO, even for *iteration count* reaching 100. Furthermore, the standard deviation of fitness fell sharply to within *stdev* $<$ 40 for *iteration count* $>$ 60. This shows that, in this experimental setting, DPSO particles had a higher rate of convergence, i.e., more particles flew to the *gbest* position than in the case of CPSO particles.

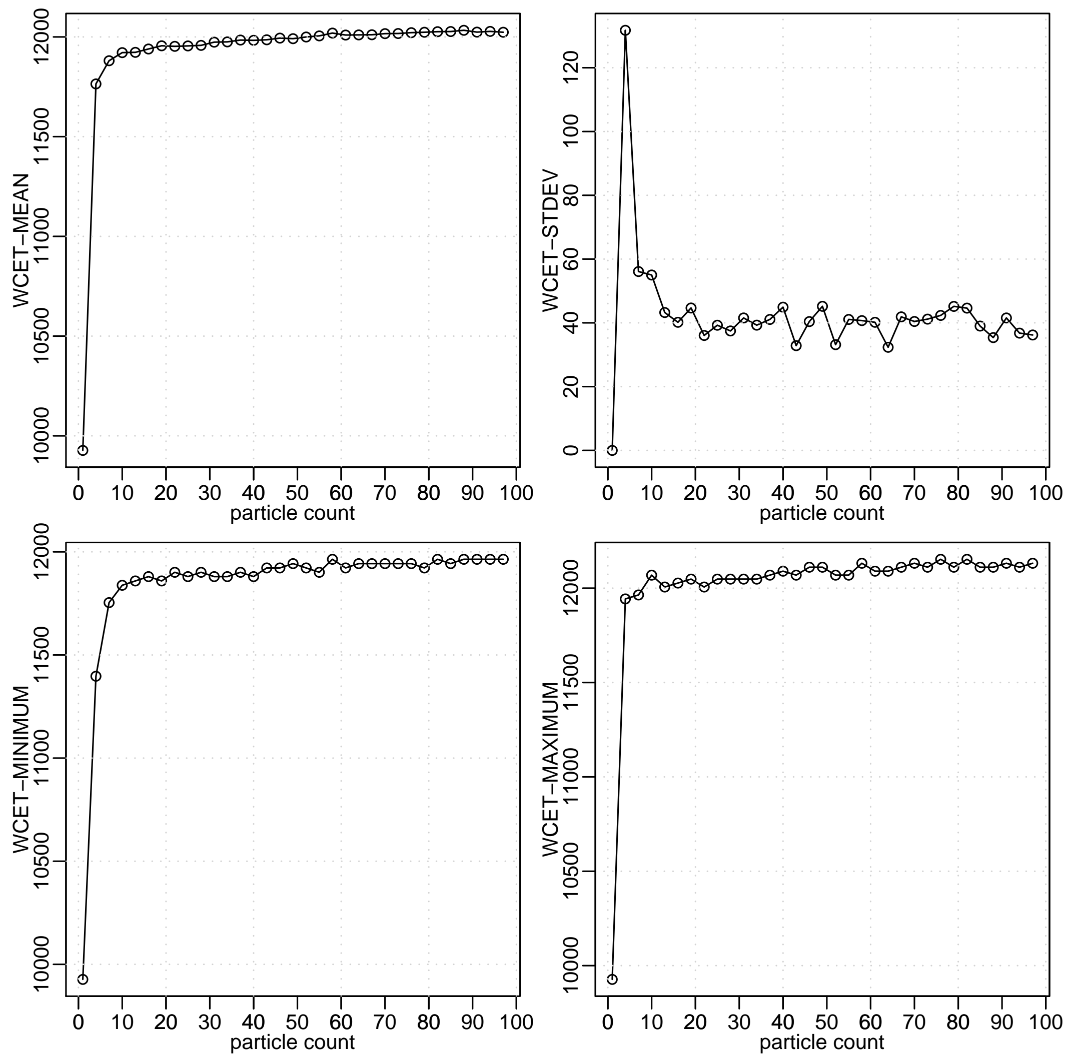

Figure 5.5: Influence of *particle count* on the WCET estimate of `bubble sort` obtained by CPSO.

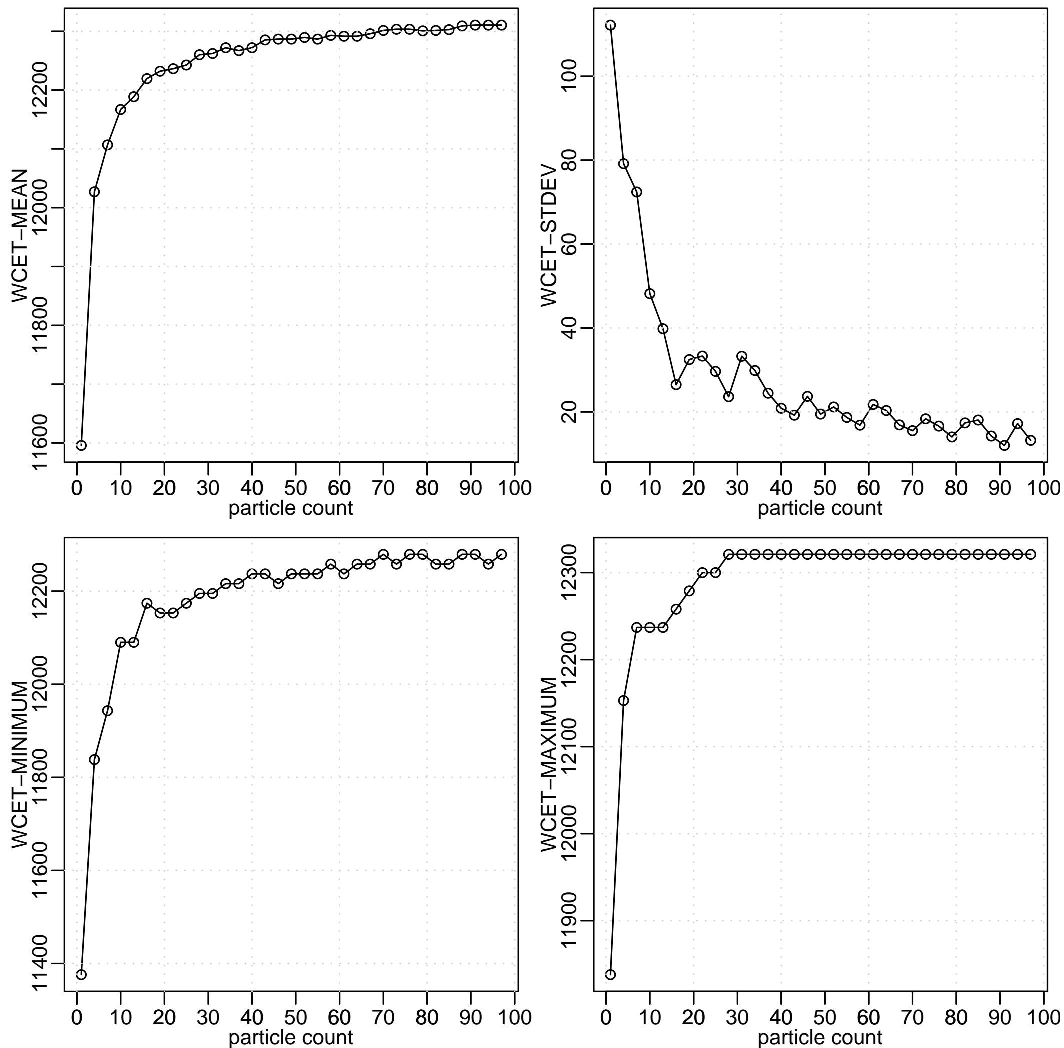

Figure 5.6: Influence of *particle count* on the WCET estimate of `bubble sort` obtained by DPSO.

5.1.3 Experiment 3: Maximum velocity and mean performance

5.1.3.1 Preparation: static calculation of WCET bounds and derivation of maximum observed execution time

A lower and upper WCET bound of the SUT `bubble sort` was statically calculated in order to obtain a measure of effectiveness for *pso_wcet*. The lower bound was necessarily too optimistic, and the upper bound was too conservative, since

the calculation employed a *greedy approach*. Greedy stands for the assumption that certain parts of code always display their worst-case behavior. The *maximum observed execution time* by *pso_wcet* lies between the calculated lower and upper WCET bound. Equations 5.1, 5.2, 5.3, and 5.5 demonstrate the application of *tree based formulas* for WCET calculation from Subsection 2.6.2 on the CFG instance of `bubble sort` shown in Figure 5.9. It was possible to apply the tree-based formulas directly on the CFG instance, since in this case, the CFG with removed cycle edges is equivalent to the syntax tree.

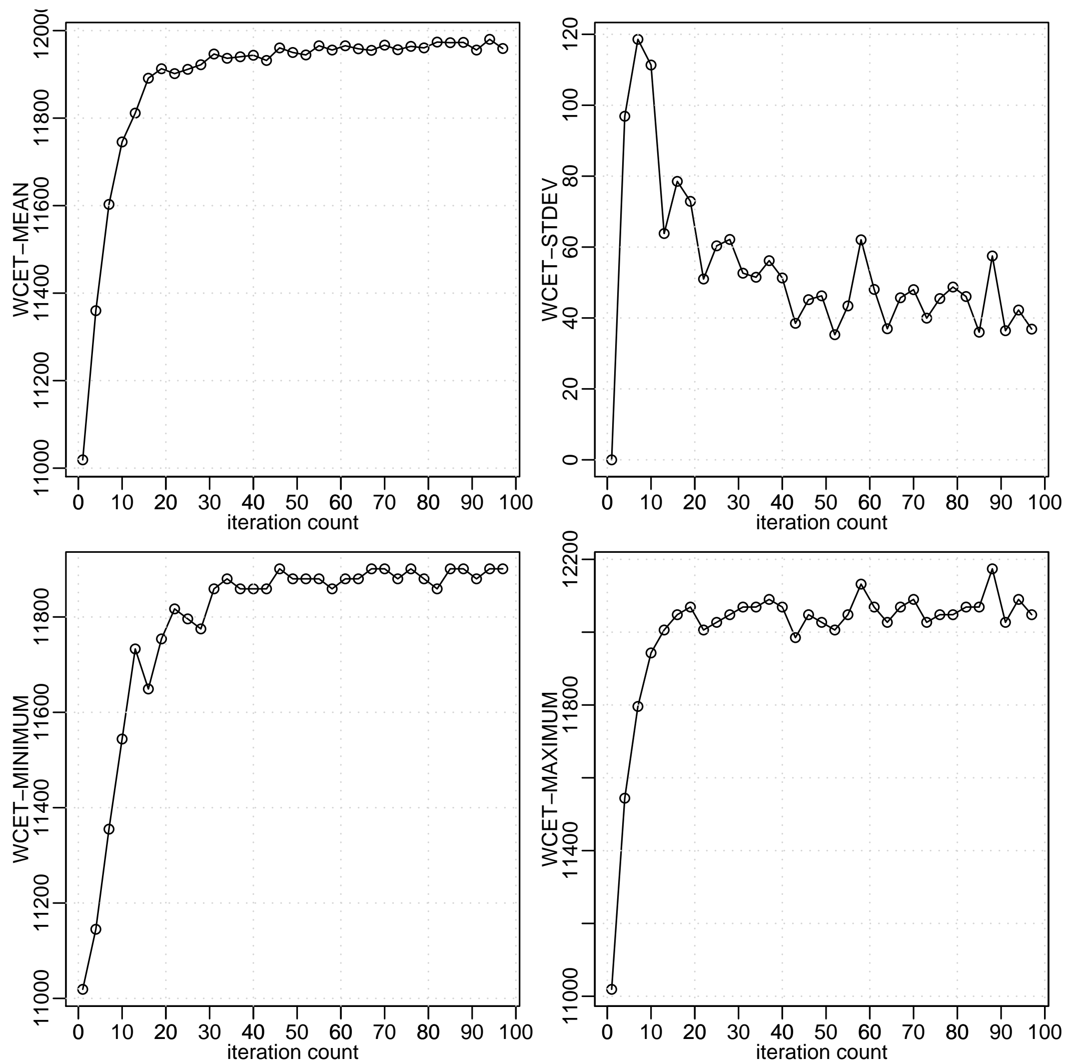

Figure 5.7: Influence of *iteration count* on the WCET estimate of `bubble sort` obtained by CPSO.

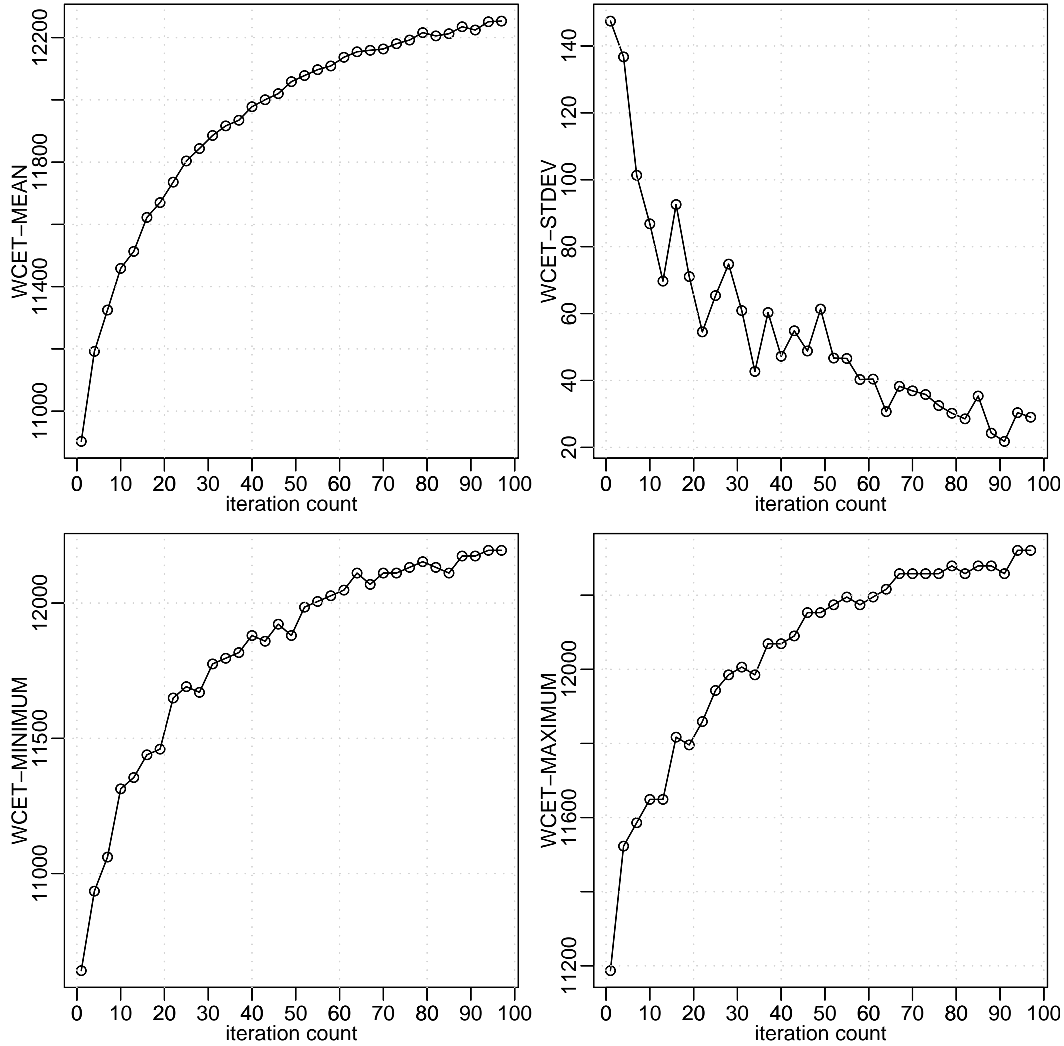

Figure 5.8: Influence of *iteration count* on the WCET estimate of `bubble sort` obtained by DPSO.

$$
\begin{aligned}
\mathit{wcet}(\mathit{bubblesort}) &= \mathit{wcet}(\mathit{entry}) + \mathit{wcet}(\mathit{LOOP}1) && (5.1)\\
\mathit{wcet}(\mathit{LOOP1}) &= \mathrm{wcet}(\mathit{for1.cond}) \\
&\quad + n\ [\mathrm{wcet}(\mathit{for1.body}) + \mathit{wcet}(\mathit{LOOP2}) + \mathrm{wcet}(\mathit{for1.inc})] \\
&\quad + \mathrm{wcet}(\mathit{for1.cond}) + \mathrm{wcet}(\mathit{for1.end}) && (5.2)\\
\mathit{wcet}(\mathit{LOOP2}) &= \mathrm{wcet}(\mathit{for2.cond}) \\
&\quad + (n-1)\,[\mathit{wcet}(\mathit{IF}) + \mathrm{wcet}(\mathit{for2.inc}) + \mathrm{wcet}(\mathit{for2.cond})] \\
&\quad + \mathrm{wcet}(\mathit{for2.end}) && (5.3)\\
& && (5.4)
\end{aligned}
$$

$$wcet(IF) = \text{wcet}(for2.body) + \max(\text{wcet}(if.then),\, 0) + \text{wcet}(if.end) \tag{5.5}$$

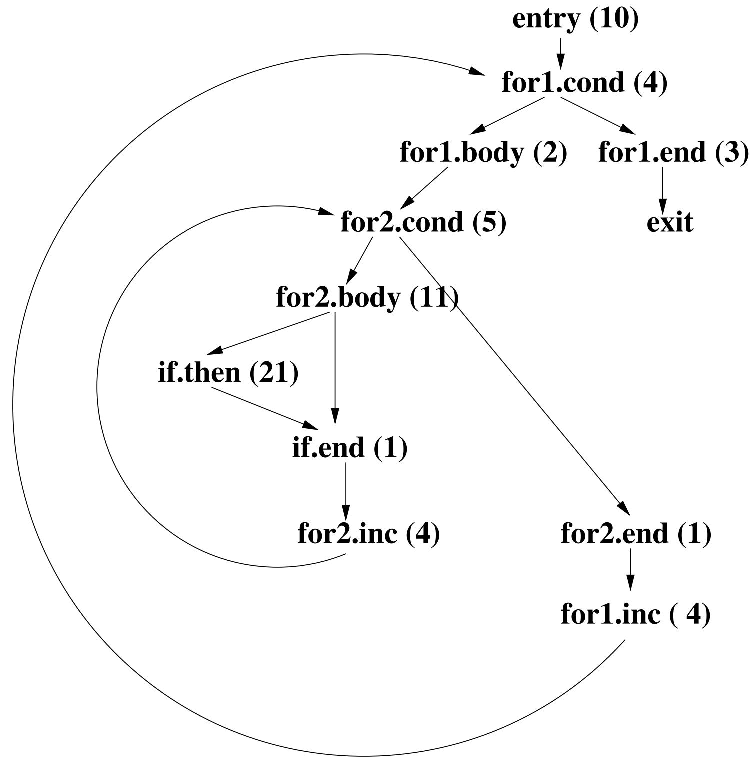

Figure 5.9: The CFG of `bubble sort` based on LLVM code that was obtained by compiling the source code in Listing 4.1. The basic blocks have their synthetic, worst-case execution times given in the parentheses. Since this CFG is almost equivalent to the program syntax tree, it can be used to calculate the WCET estimate with tree-based formulas from Subsection 2.6.2.

Using the SUT configuration from Table 5.2, the *loop bound* n in Equations 5.2 and 5.3 was substituted with the size of the *SUT input vector*. Using this information and the WCET value of each basic block, given in Figure 5.9, it was possible to calculate the WCET bound of the whole program. The upper WCET bound was obtained by taking the maximal execution time of the *IF1* alternative as specified in Equation 5.5. To obtain the lower WCET bound, a minimum instead of the maximum was used in the same equation.

The cause for underestimation by the lower bound is that the WCET inducing execution path does not always pass through the minimal sequence of basic blocks: *for2.body* → *if.end*, for each loop iteration. The cause for overestimation by the upper bound is that the WCET path neither traverses the maximal sequence:

	Time units	Error (%)
Lower static WCET bound	8317	32.5
Maximum observed and real WCET	12321	-
Upper static WCET bound	16297	32.37

Table 5.6: The lower and upper WCET bound and the maximum observed and real WCET of `bubble sort`. The lower and upper bound were calculated by applying the tree-based formulas as shown in Equations 5.1 - 5.5. The real WCET was obtained by using the a priori known input data that exercise the longest execution path in bubble sort. The maximum observed WCET, which in this case corresponds to the real WCET, was obtained empirically by *pso_wcet* using extensive testing. Even for a small program, the lower and upper bound obtained by the tree-based formulas exhibited significant underestimation, viz., overestimation of the real WCET.

for2.body → *if.then* → *if.end*, for each loop iteration. In some iterations, the WCET inducing path passes through the minimal sequence and in others through the maximal sequence.

The worst-case inducing input vector for `bubble sort` is known from traditional algorithmic analysis to be an *inversely sorted list*. Extensive testing by *pso_wcet* also confirmed this.

The WCET bounds calculated by the tree-based formulas and the *maximum observed execution time* obtained by *pso_wcet* are assembled in Table 5.6. The maximum observed execution time was used as the *termination condition* for the following experiment.

5.1.3.2 Settings

The goal of this experiment was to examine the effects of the PSO parameters on the mean *iteration count* necessary for finding the optimum solution, i.e., the WCET inducing input vector.

The selection of parameters was based on the results of preceding two experiments. The chosen value-pairs for the cognitive and social constant (c_1, c_2) are assembled in Table 5.7. The rationale used for the selection of c_1 and c_2 from a high number of possible combinations was based on the experimental results presented in Subsection 5.1.2.1. In the current experiment, a selection of (c_1, c_2) pairs was tested over a range of maximum velocity values V_{max}.

The experiment was carried out using the SUT configuration assembled in Table 5.2, the values for topology/metric/encoding from Table 5.5, and the specific run-configuration of the optimizer from Table 5.8.

Algorithm	c_1	c_2
CPSO	2	2
	1	1
	1.2	1
	1.4	1
DPSO	1	1
	1.5	1.5
	1.9	1.7
	0	2

Table 5.7: Pairs of cognitive and social constants (c_1, c_2) used in Experiment 3 on the SUT `bubble sort`. The selection of values was based on the results of Experiment 2 discussed in Subsection 5.2.2.

Particle count for CPSO	30
Particle count for DPSO	50
Termination condition	fitness = maximum observed WCET or iteration count reached 1000
Runs	100

Table 5.8: Configuration of Experiment 3 on `bubble sort`.

The *particle count* in Table 5.8 was chosen based on the results of the preceding experiment that were documented in Subsection 5.1.2.3. For each algorithm, a *particle count* with high mean fitness but below the *particle utility limit* was selected.

The *termination condition* in Table 5.8 was specified in such a way that each run instance of the experiment should terminate when it finds the optimum solution or when it reaches the maximum number of allowed iterations. A comparatively high number of allowed iterations was selected in order to distinguish the case of failure to find the WCET from the case of suboptimal performance.

5.1.3.3 Results

The results of Experiment 3 on `bubble sort` are plotted in Figure 5.10. For the CPSO algorithm, only two maximum velocity values: $V_{max} = 1$ and $V_{max} = 2$ out of the total of 16 tested values guided the optimization successfully to the optimum solution. In contrast, the range of successful V_{max} values for the DPSO algorithm was much larger: $4 \leq V_{max} \leq 10$.

Figure 5.10 shows that V_{max} had more influence than the cognitive and social

constant c_1, c_2 on the performance of both CPSO and DPSO. Optimization runs in this experiment failed only due to incorrect V_{max} settings. The effect of different $(c_1,\ c_2)$ settings was limited to small quantitative differences in the number of iterations needed to find the optimum.

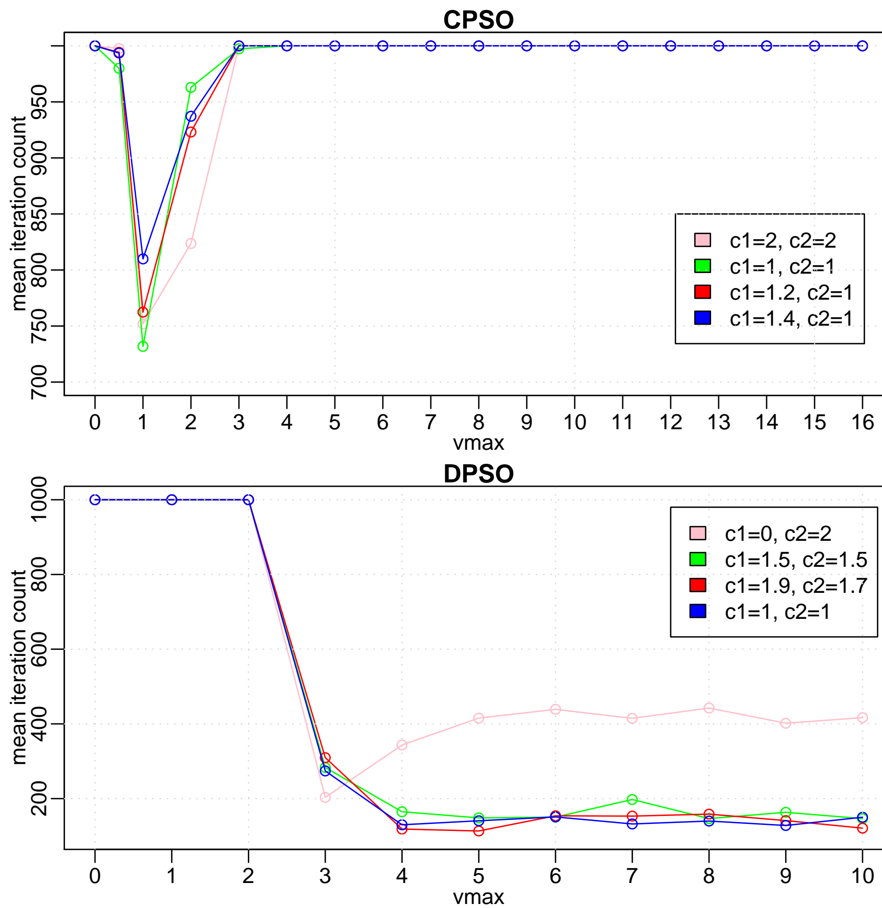

Figure 5.10: Influence of maximum particle velocity, V_{max}, in connection with a selection of learning factors $(c_1,\ c_2)$ on the mean iteration count needed to find the WCET inducing solution of `bubble sort`. A mean value of 1000 for a given set of parameters c_1, c_2 and V_{max} indicates that the PSO algorithm did not find the WCET solution in any of the run instances within allocated computational resources. It can be seen from the second graph that DPSO managed to find the optimum solution even for the cognitive constant $c_1 = 0$, i.e., by particles doing only group and no individual search.

5.1.3.4 Interpretation of results

In an overview work on PSO, [41], it is stated that for CPSO: 'no definite conclusions about the asymmetric learning factors (c_1, c_2) have been reported'. In comparison, the optimum for CPSO in Figure 5.10 was achieved by symmetric learning factors (c_1=1, c_2=1). For DPSO in the same Figure, the asymmetric factors (c_1= 1.9, c_2= 1.7) slightly outperformed the symmetric factors ($c_1 = 1$, $c_2 = 1$).

In [54], it is reported that the cognitive and social constant are both essential to the success of CPSO. In contrast, DPSO succeeded in the current experiment even without the cognitive constant, i.e., with the setting ($c_1 = 0$, $c_2 = 2$). However, the algorithm's quantitative performance was handicapped when compared to the setting of $c_1 > 0$. The inverse case ($c_1 > 0$, $c_2 = 0$), i.e., particles conducting only individual search, would be unlikely to succeed, based already on the results of the previous experiment which are shown in Figure 5.2.

In [41], it is stated that for CPSO, V_{max} is generally set to the value of the dynamic range of each variable, i.e., to X_{max}. In contrast, Figure 5.10 shows that CPSO clearly failed for this proposed setting, $V_{max} = X_{max} = 16$, and for a whole range of other unsuitable settings $V_{max} \in [3, X_{max}]$. A set of random sample experiments, not documented here, was conducted in order to explain this strong influence of V_{max} on the performace of CPSO. Based on the obtained results, there is reason to hypothesize that the optimum V_{max} setting depends on the size of the search space. Furthermore, it may depend on the granularity of the evaluated search space coordinates. In the case of `bubble sort`, CPSO particles move in a continuous space, but their coordinates have to be rounded to the nearest integer before they are used as an input vector for the SUT and evaluated. This may explain why CPSO found the optimum only for values $V_{max} = 1$ and $V_{max} = 2$ in the current experiment.

To conclude, in Experiment 3, in which fine-tuned PSO parameters were used, the best case mean performance of DPSO was four times better than that of CPSO. Both of the algorithms succeeded in finding the optimum.

5.2 Optimization of Rastrigin's Function

The second set of experiments designed to evaluate the *pso_wcet* framework deals with the optimization of `Rastrigin's` function [55], a function with many local optima that is often used to test the performance of optimization algorithms. This function belongs to one of the more difficult benchmarks and is well established in literature [45, 44, 39].

In contrast to the discrete problem of finding the WCET of `bubble sort`, for which there are multiple optimum solutions, the Rastrigin's function is continuous

and has only a single optimum. The optimal solutions, i.e., the optimal coordinates in search space are much more sparse for the Rastrigin's function than for the problem of maximizing the WCET estimate of `bubble sort`. The main difficulty in optimizing the Rastrigin's function are its many *local optima*. Genetic algorithms, as well as some PSO implementations can get stuck at one of these positions, viz., all population entities may converge to a single local optimum.

A 2-dimensional Rastrigin's function with the minimum at coordinates $(0,0)$ is displayed in Figure 5.11. The configuration of the optimization problem that is used throughout the following experiments is assembled in Table 5.9. The task of the evaluated algorithms: *random search*, CPSO and DPSO was to find the global optimum by progressive function minimization.

Software under test	`Rastrigin`
Computation result	$f(\vec{x}) = \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i) + 10), \quad [n = 2]$
Fitness function	computation result
Optimization type	minimization
Type of input vector	[`double`]
Size of input vector	2
Search space	$[-256, 256]^2$

Table 5.9: Configuration of the optimization problem `Rastrigin`.

5.2.1 Experiment 1: Effect of topology, metric, and encoding

The goal of the first experiment on function `Rastrigin` was to survey different combinations of topology, metric, and encoding and to compare their effect on the performance of *random search*, CPSO, and DPSO. As a result, the best combination of parameters was determined for each algorithm. For CPSO this was the *star topology* with *initial metric*, while DPSO performed best using the *circle topology*, *fitness based metric*, and *gray encoding*.

PSO particles	50
Iterations per run	1000
Total runs	100

Table 5.10: General configuration of PSO for experiments on `Rastrigin`.

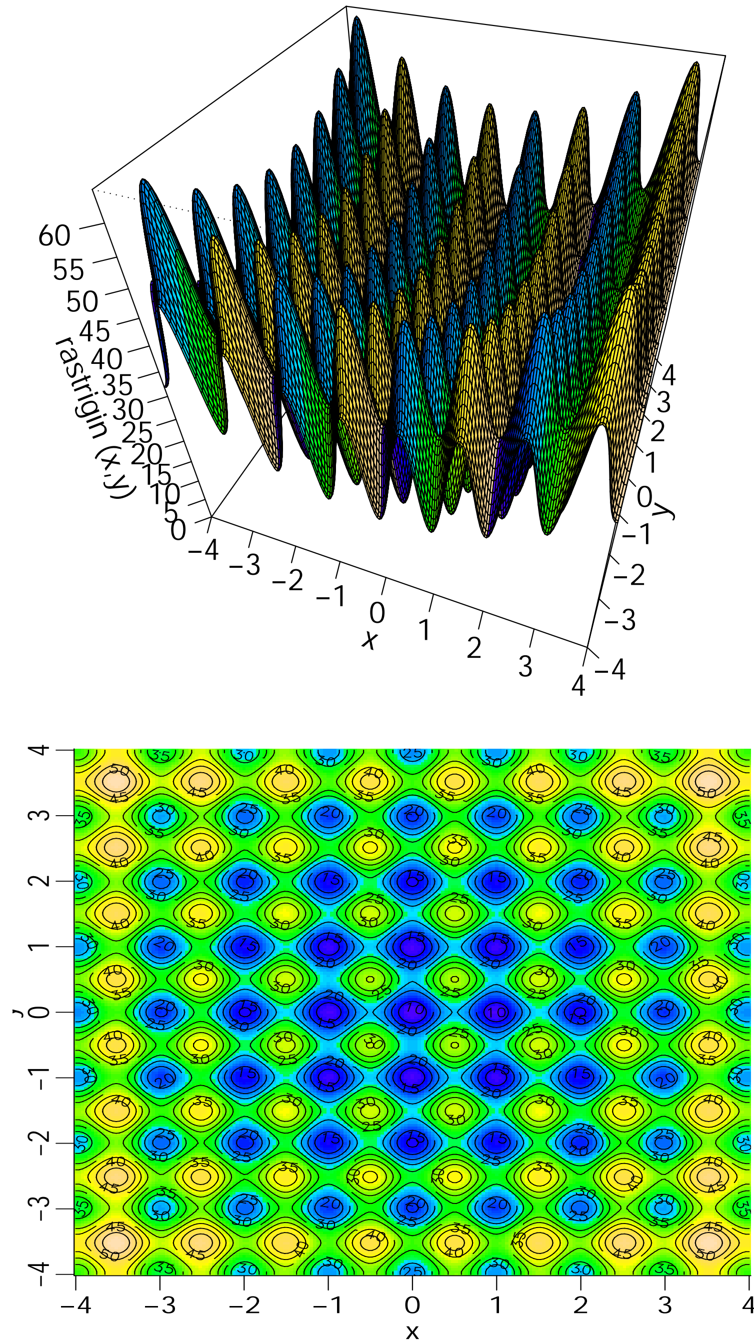

Figure 5.11: The `Rastrigin's` function belongs to one of the more challenging benchmarks for evaluating heuristic algorithms. The above graphs visualize the function for the case $n = 2$ of formula: $f(\vec{x}) = \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i) + 10)$. In contrast to the *discrete problem* of maximizing the execution time of `bubble sort`, which has many optimal solutions, `Rastrigin's` function is continuous and has only one global optimum at coordinates $(0, \ldots, 0)$. The difficulty of optimizing this function by heuristic algorithms arises from the fact that it has many local optima.

Parameter	CPSO	DPSO
Cognitive constant c_1	1	1.9
Social constant c_2	1	1.7
Maximum velocity limit V_{max}	1	4
Granularity of particle movement	-	0.01
Zero speed outside search space	True	-

Table 5.11: Default configuration of PSO for experiments on `Rastrigin`.

5.2.1.1 Settings

The configuration of the optimization problem `Rastrigin` that was used in Experiment 1 is displayed in Table 5.9. The employed PSO settings are assembled in Tables 5.10 and 5.11.

5.2.1.2 Results

The results of Experiment 1 are assembled in Table 5.12. Both PSO algorithms performed much better than random search. In terms of mean fitness CPSO clearly outperformed DPSO. This is different from Experiment 1 on `bubble sort`, where DPSO always outperformed CPSO in terms of both mean and extreme fitness. From this experiment, however, it can be observed that CPSO had difficulties in fine-tuning the solution once particles were in the region of the global optimum; CPSO came close, but did not succeed in finding the exact optimum. This might be due to a faulty setting for maximum velocity V_{max} and suboptimal settings for the cognitive and social constants c_1, c_2. The optimal parameterization for each algorithm is determined later in Experiments 2 and 3.

The best combination of topology/metric for CPSO was the star/initial combination, which is the same conclusion as in Experiment 1 on `bubble sort`. For the DPSO algorithm, the circle/fitness combination provided the best results; the wheel topology provided the second best results after the circle topology. This is a surprise, since in Experiment 1 on `bubble sort`, circle and wheel topologies induced worse performance of DPSO than the star topology.

A comparison of encoding for DPSO shows that gray encoding was overall superior to binary encoding for the star topology, while binary encoding was better than the corresponding gray encoding for the wheel topology. Inconclusive results for encoding were obtained with the circle topology.

Based on these observations, an optimal combination of topology, metric and encoding, was selected for CPSO and DPSO and used in Experiments 2 and 3. It is assembled in Table 5.13.

Experiment 1: Minimization of Rastrigin's function								
Algorithm	Topology	Metric	Encoding	Minimum	Maximum	Mean	Stdev	Mean rank
Random search	-	-	binary	*3.443166*	*31.201209*	*15.085969*	**6.176458**	*2*
	-	-	gray	**2.079086**	**28.568722**	**14.972588**	*6.656456*	**1**
CPSO	star	initial	-	0.000097	**0.026250**	**0.008576**	**0.006561**	**1**
	wheel	initial	-	*0.000022*	0.083992	0.015236	0.015412	5
	circle	initial	-	0.000035	0.083309	0.015772	0.016228	6
	circle	fitness	-	**0.000015**	0.043529	*0.010014*	*0.009278*	*2*
	circle	euclid	-	0.000106	0.053682	0.013499	0.011478	4
	circle	taxi	-	0.000107	*0.042642*	0.011837	0.010729	3
DPSO	star	initial	binary	**0.000000**	31.876055	8.439318	9.207519	6b
	star	initial	gray	**0.000000**	31.857280	7.815162	8.078447	6a
	wheel	initial	binary	**0.000000**	4.157860	0.655858	0.938501	2a
	wheel	initial	gray	**0.000000**	15.998506	1.192862	2.828908	5b
	circle	initial	binary	**0.000000**	5.019086	0.839589	0.887523	3a
	circle	initial	gray	**0.000000**	15.919253	0.794892	1.637172	2b
	circle	fitness	binary	**0.000000**	*2.357280*	*0.612021*	*0.581065*	*1b*
	circle	fitness	gray	**0.000000**	**1.999665**	**0.529161**	**0.556208**	**1a**
	circle	euclid	binary	**0.000000**	5.099086	0.933366	0.950241	4a
	circle	euclid	gray	**0.000000**	5.038506	0.973018	0.914960	4b
	circle	taxi	binary	**0.000000**	5.057860	0.853111	0.887358	3b
	circle	taxi	gray	**0.000000**	8.656649	1.176241	1.236614	5a

Table 5.12: Different combinations of topology/metric/encoding were investigated for their effect on the performance of *random search*, CPSO, and DPSO. For each algorithm, the entries in bold are the best and the entries in italics are the second best of each column.

Algorithm	Topology	Metric	Encoding
CPSO	star	initial	-
DPSO	circle	fitness	gray

Table 5.13: Configuration of PSO for the optimization problem `Rastrigin` in Experiments 2 and 3. The combination of topology, metric, and encoding with optimum performance in Experiment 1 was chosen for further application in these experiments.

5.2.2 Experiment 2: Effect of PSO learning factors

The second experiment dealt with the effect of pairs of cognitive and social constants (c_1, c_2) on the performance of CPSO and DPSO. The objective was to obtain a set of pairs (c_1, c_2) that induce good performance; a set of pairs with interesting properties was later used for evaluating the settings for maximum velocity V_{max} in Experiment 3. The pair combinations (c_1, c_2), evaluated in this experiment, were selected from the spaces $[0, 2]^2$ and $[1, 3]^2$ for CPSO and DPSO respectively. As a result of the experiment, regions with suboptimal performance in the (c_1, c_2) space were identified.

5.2.2.1 Settings

The configuration of the optimization problem `Rastrigin` used in this experiment is displayed in Table 5.9. The resources allocated to the particle swarm optimizer and the number of experiment runs are assembled in Table 5.10. The selected topology, metric, and encoding, as well as the maximum velocity V_{max} are displayed in Tables 5.13 and 5.11 respectively.

5.2.2.2 Results

The results of the experiment for CPSO are shown in Figure 5.12. One can observe that all (c_1, c_2) value pairs, except those with social constant $c_2 = 0$, induced the same mean and extreme case performance. For the setting of social constant $c_2 = 0$, optimization failed. There were differences in the standard deviation of fitness for different (c_1, c_2) pairs: the standard deviation increased in the direction of $(c_1, c_2) \rightarrow (0.0, 0.2)$.

The results for DPSO are displayed in Figure 5.13. The function was successfully minimized for all evaluated pairs (c_1, c_2). The best case fitness, i.e. the minimum of each experiment run was zero. The mean and worst-case fitness was optimal in the region around $(c_1, c_2) = (3.0, 1.0)$ and worsened in the

direction of $(c_1, c_2) \rightarrow (1, 3)$. The standard deviation increased in the direction: $(c_1, c_2) \rightarrow (1.0, 3.0)$.

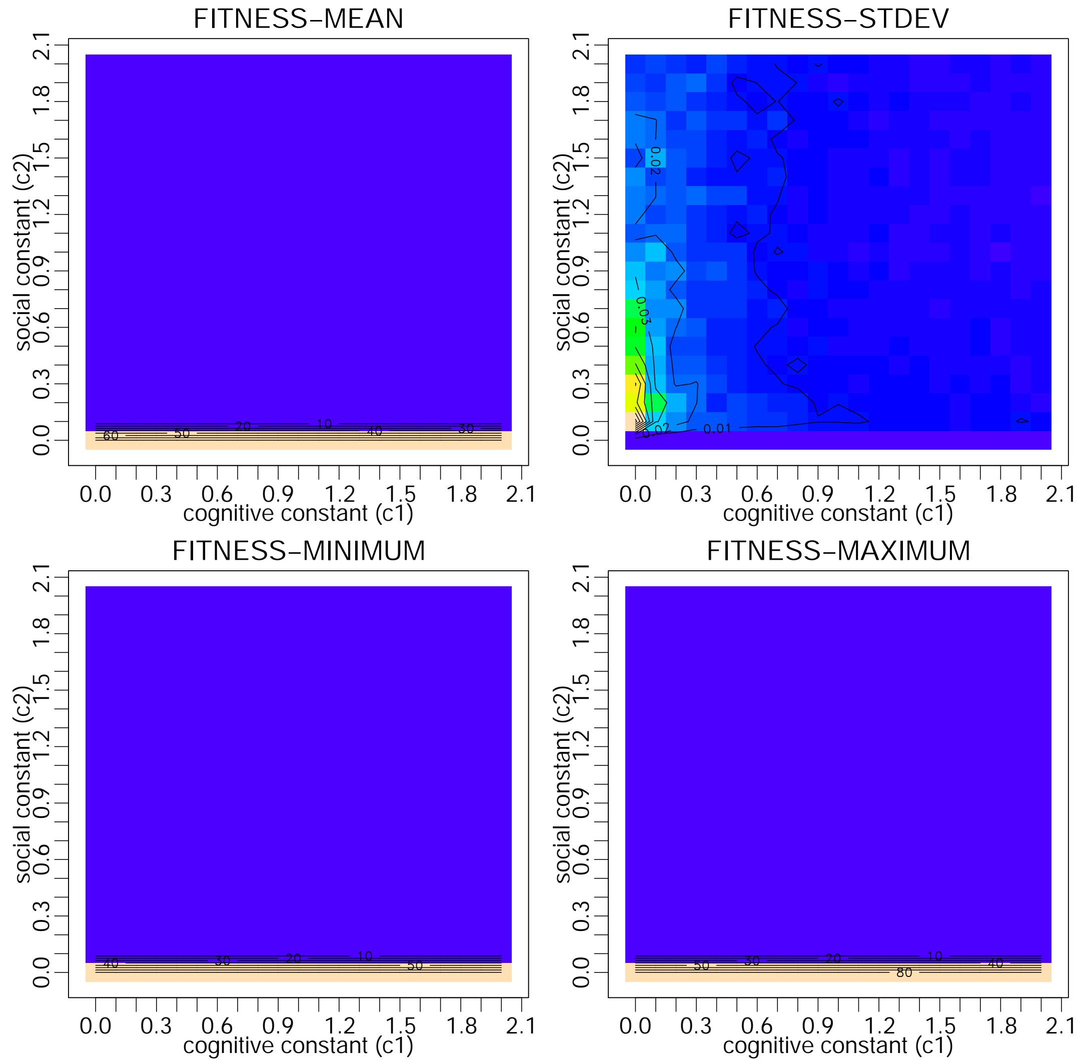

Figure 5.12: Algorithm CPSO - Influence of the cognitive and social constant $(c_1,\, c_2)$ on the minimization of function `Rastrigin`. The function was successfully minimized for all $(c_1,\, c_2)$ combinations except those with $c_2 = 0.0$.

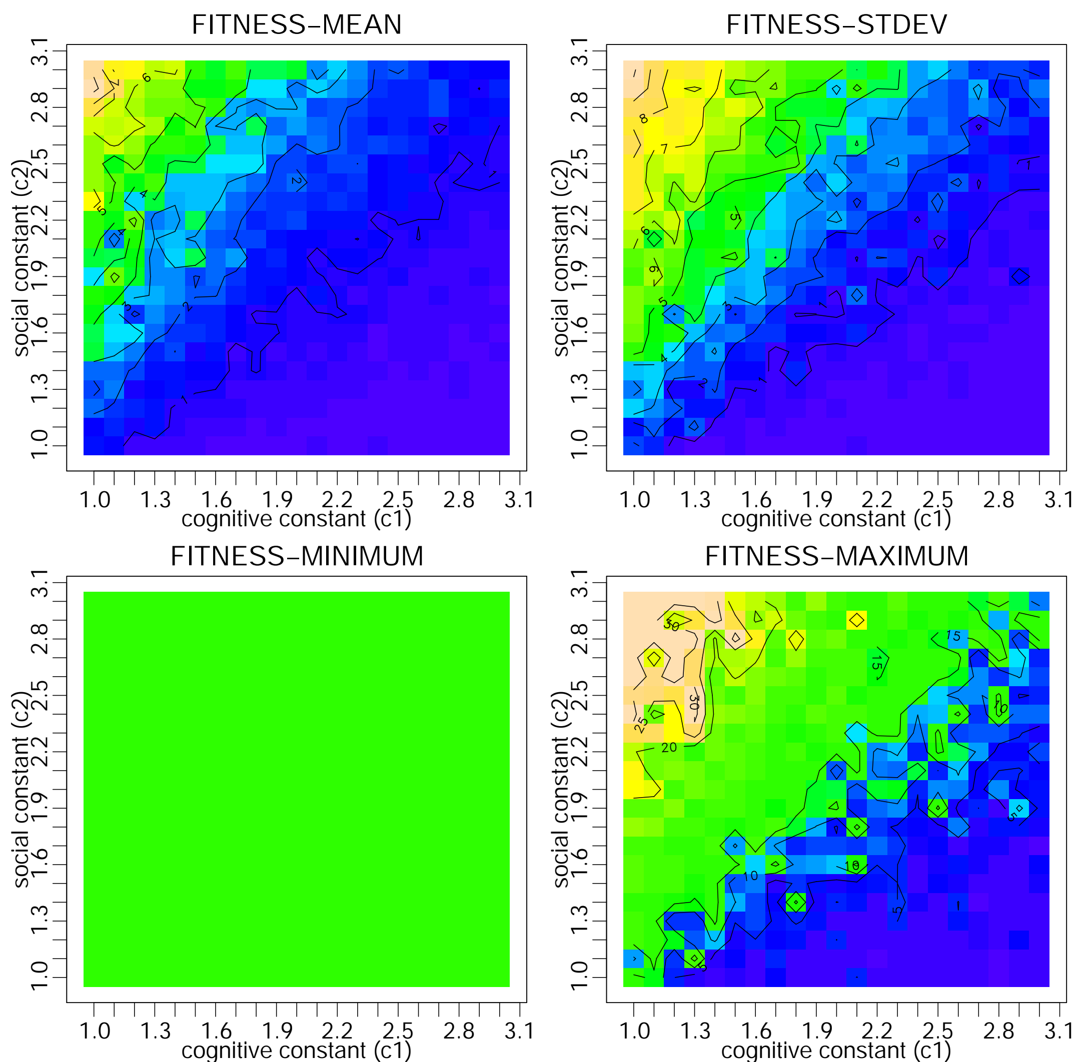

Figure 5.13: Algorithm DPSO - Influence of the cognitive and social constant (c_1, c_2) on the minimization of `Rastrigin's` function. There was strong influence on mean and worst-case (maximum) performance, while the best-case performance (minimum) was the same for all (c_1, c_2) combinations.

5.2.3 Experiment 3: Maximum velocity and mean iteration count

The goal of Experiment 3 was to survey a range of different values for maximum velocity V_{max} and to determine the effect on performance. A set of learning factors (c_1, c_2), assembled in Table 5.14, was used with different V_{max} settings. The selec-

tion of learning factors was based on the results of Experiment 2. In the current experiment, reasonable performance of CPSO was achieved for $V_{max} \in [0.5, 3]$, while DPSO performed well for $V_{max} \in [3, 10]$.

5.2.3.1 Settings

The same settings as in Experiment 2 were used for the optimization problem configuration, shown in Table 5.9, and for the choice of topology, metric and encoding, shown in Table 5.13. New settings included the pairs of learning factors which are assembled in Table 5.14.

The run settings of the experiment are displayed in Table 5.15. The interpretation of the termination condition in this table is the following: if a solution is found whose fitness is inside the interval $[0, 0.01]$, the run instance terminates and the iteration count necessary to achieve that solution is recorded. Otherwise, if the run instance iterates up to 2000 iterations, it is assumed that it failed to reach the solution within the allocated resource constraints. In this context, the resource constraints are the particle count and the allowed iteration count.

Algorithm	c_1	c_2
CPSO	2	2
	0.1	0.1
	1	1
	0	2
DPSO	3	1
	2.5	1.5
	3	3
	1.1	2.9

Table 5.14: Pairs of cognitive and social constants (c_1, c_2) used in Experiment 3 on `Rastrigin`. The selection of values was based on Experiment 2 and was discussed in Subsection 5.2.2.

5.2.3.2 Results

The two graphs in Figure 5.14 display the mean number of iterations of CPSO and DPSO, for different pairs of cognitive and social constant (c_1, c_2), as a function of maximum velocity V_{max}.

For the CPSO algorithm, the lowest mean number of iterations was achieved by the value $V_{max} = 0.5$, for all combinations (c_1, c_2). This result is consistent with

Particle count for CPSO	50
Particle count for DPSO	50
Termination condition	$fitness \leq 0.01$ or iteration count reached 2000
Runs	100

Table 5.15: Special configuration of PSO used in Experiment 3 on `Rastrigin's` function.

Experiment 3 on `bubble sort` where $V_{max} = 1$ was optimal. Among the tested (c_1, c_2) pairs, the pair $(c_1, c_2) = (2, 2)$ was optimal for CPSO.

The DPSO algorithm succeeded in finding the optimum region for all values $V_{max} \geq 3$. The lowest and optimal iteration count was recorded for $V_{max} = 4$ and $(c_1, c_2) = (1.1, 2.9)$. These results of DPSO are consistent with those of Experiment 3 on `bubble sort`, where the success in finding the optimum was also recorded starting from $V_{max} \geq 3$.

Another observation regarding the CPSO algorithm confirms the hypothesis, introduced in Subsection 5.1.3.4, that a successful V_{max} setting is dependent on the size of the search space. In Experiment 3 on bubble sort, successful V_{max} values were inside the interval $[1.0, 2.0]$. That experiment had a smaller search space than the one with Rastrigin, i.e., the dimensional search range was $[-16, 15]$ as opposed to $[-256, 256]$. In Experiment 3 on Rastrigin, successful V_{max} values were found in a larger interval: $[0.5, 16]$. The difference in size of the dimensional range was sixteen times, while the difference in size of the successful V_{max} interval was about eight times.

5.3 Summary

A series of experiments was conducted on algorithms: CPSO, DPSO, and random search. The evaluated problems were maximization of the WCET estimate of bubble sort and optimization of the ***`Rastrigin's`** function*. The former problem is discrete while the latter is continuous; each one presented a challenge to optimization in its own way. Furthermore, a systematic approach to selecting and fine-tuning of PSO parameters for a specific problem was presented.

The two PSO algorithms outperformed random search by a large margin in all experiments. After suitable parameters had been selected, both PSO algorithms successfully found the optimum of each optimization problem in a reasonable amount of computation time. The conclusion from literature [41], that CPSO has difficulties in dealing with discrete variables, was confirmed experimentally.

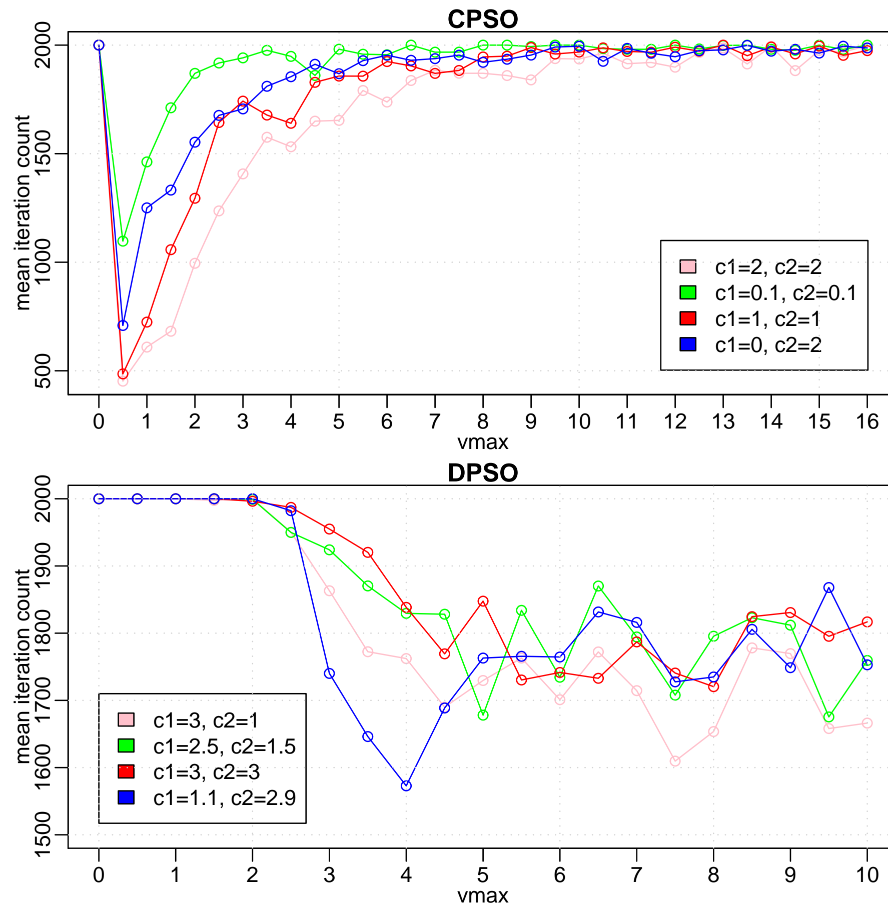

Figure 5.14: Influence of maximum particle velocity, V_{max}, in connection with a selection of cognitive-social pairs (c_1, c_2) on the mean iteration count necessary to find the optimum for `Rastrigin`. A value of 2000 for the mean iteration count indicates that a given PSO algorithm did not succeed in reaching the region of space, specified around the optimum, in any of the run instances.

The performance of DPSO was superior to that of CPSO on `bubble sort`. On the continuous problem - `Rastrigin's` function - DPSO had better extreme case performance, while CPSO had better mean performance.

For the case of maximization of the WCET estimate of bubble sort, the WCET estimate provided by PSO was much tighter then the upper bound calculated by *tree-based formulas*. The input data generated by PSO that induced the maximum WCET estimate were identical to the worst-case input of `bubble sort` known from traditional algorithm theory.

Furthermore, it was found that the choice of *topology* can have significant influence on the performance of both PSO algorithms. For the CPSO algorithm, the *star topology* provided optimal performance on both optimization problems. For the DPSO algorithm on the `bubble sort` problem, the star topology was also optimal, while the *circle topology* provided optimal performance on the `Rastrigin` problem. A further increase in DPSO performance on `Rastrigin` was achieved by combining the circle topology with the *fitness distance metric*. However, this increase was relatively small when compared to the increase provided by the circle topology alone. It would seem that the right choice of topology has more influence on PSO performance than the right choice of *metric*. Even the simple *initial metric* worked comparatively well and outperformed more complex metrics in a few cases.

Experiments also showed that the encoding of particle coordinates can have great influence on the performance of DPSO; *gray encoding* was better than *binary encoding* in most of the cases.

Among the evaluated pairs of *cognitive and social constants* (c_1, c_2), the pair $(c_1, c_2) = (2, 2)$ provided the most efficient optimization of both test problems for CPSO. For DPSO, a number of pairs $(c_1, c_2) \in [1, 3]^2$ proved to be efficient. *Asymmetric pairs*, $c_1 \neq c_2$, were not significantly worse than the *symmetric pairs* $c_1 = c_2$. The only limitation on the setting of the cognitive and social constant, observed for both PSO algorithms, was that the social constant should always be greater than zero, i.e., $c_2 > 0$. Furthermore, it was found that search with the zero cognitive constant, $c_1 = 0$, works, but is less efficient than a higher than zero setting.

As far as the maximum velocity V_{max} is concerned, it was found that the correct setting of this parameter is much more important than the fine-tuning of cognitive-social pairs (c_1, c_2). A faulty setting for V_{max} can have as its consequence the failure of the particle swarm to converge upon the optimum solution, even after the swarm discovers the region of space in its vicinity. For CPSO, search with maximum velocity values, $V_{max} \in [0.5, 3]$, was successful on both optimization problems. For DPSO, search with values, $V_{max} \in [3, 10]$, was likewise successful.

It was observed that a linear increase in the size of the particle swarm above a certain value - the *particle utility limit* - did not result in the corresponding

increase in performance. In terms of performance, the computational effort was usually better spent on a higher number of algorithm iterations.

The observations obtained from these experiments might be useful in parameterizing PSO for other optimization problems.

Chapter 6

Conclusion

The question which this work attempted to answer was how to apply a relatively novel, heuristic optimization technique - particle swarm optimization - on the problem of deriving worst-case execution time. The preparatory part of the work contains an overview of different methods for WCET analysis (static, measurement-based, hybrid) and an analysis of two representative PSO algorithms.

The continuous and discrete binary PSO were implemented and integrated in a framework for synthetic WCET analysis. The framework delivers a lower WCET bound by maximizing the execution path length. The maximization is achieved by progressively optimizing the input data which are used to exercise the SUT.

A series of detailed experiments concerning the right parameterization of PSO was performed. It was shown that PSO is a flexible optimization method; it worked satisfactorily with default settings (out of the box), although performance could be improved by fine-tuning. Of all the parameters, it was determined that maximum velocity, V_{max}, has the most potential for inducing a failure of optimization.

The optimally parameterized PSO was used to determine the longest execution path of `bubble sort`. The quality of the implementation and parameterization were such, that the derived lower bound was equal to the analytically obtained WCET. In comparison, WCET bounds calculated by static tree-based formulas had an error margin of 30%. The difference between the lower bounds obtained by PSO and random search using the same computational resources was 525 time units.

The framework was also adapted for white-box testing by introducing additional fitness functions. The optimization goals can be specified as execution path targets. Thus, paths can be empirically checked for feasibility. Furthermore, input data exercising specific SUT regions can be generated, which can be useful in measurement-based WCET analysis.

Bibliography

[1] R. Kirner, W. Zimmermann, and D. Richter, "On undecidability results of real programming languages," in *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, (Maria Taferl, Austria), Oct. 2009.

[2] A. Colin and I. Puaut, "A modular & retargetable framework for tree-based wcet analysis," *Real-Time Systems, Euromicro Conference on*, vol. 0, p. 0037, 2001.

[3] W. Stallings, *Operating Systems (5th Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[4] G. Bernat, A. Colin, and S. M. Petters, "Wcet analysis of probabilistic hard real-time systems," *Real-Time Systems Symposium, IEEE International*, vol. 0, p. 279, 2002.

[5] P. Puschner, "Experiments with wcet-oriented programming and the single-path architecture," in *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 205–210, Feb. 2005.

[6] M. Ltd, "MISRA-C:2004 Guidelines for the use of the C language in critical systems," Oct. 2004.

[7] D. Barkah, A. Ermedahl, J. Gustafsson, B. Lisper, and C. Sandberg, "Evaluation of automatic flow analysis for wcet calculation on industrial real-time system code," in *20th Euromicro Conference of Real-Time Systems, (ECRTS08)*, July 2008.

[8] R. Kirner and P. P. Puschner, "Classification of WCET analysis techniques," in *ISORC*, pp. 190–199, IEEE Computer Society, 2005.

[9] R. Kirner, *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[10] C. Cullmann and F. Martin, "Data-flow based detection of loop bounds," in *WCET*, 2007.

[11] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *RTSS*, pp. 57–66, 2006.

[12] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra, "Exploiting branch constraints without exhaustive path enumeration," in *WCET*, 2005.

[13] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *STTT*, vol. 4, no. 4, pp. 437–455, 2003.

[14] A. Kadlec, R. Kirner, and P. Puschner, "Avoiding timing anomalies using code transformations," *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, vol. 0, pp. 123–132, 2010.

[15] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, 2001.

[17] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[18] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," pp. 12 –21, 1999.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[20] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *WCET*, 2006.

[21] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *J. Syst. Archit.*, vol. 46, no. 4, pp. 339–355, 2000.

[22] G. Berry, "The effectiveness of synchronous languages for the development of safety-critical systems," 2003.

[23] S. Ramesh and P. Sampath, *Next generation design and verification methodologies for distributed embedded control systems: proceedings of the GM R&D Workshop, Bangalore, India, January 2007*. Springer, 2007.

[24] P. Puschner and C. Koza, "Calculating the maximum, execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[25] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," in *Intelligent Systems at the Service of Mankind*, vol. 2, UBooks Verlag, Dec. 2005.

[26] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *SIGPLAN Not.*, vol. 30, no. 11, pp. 88–98, 1995.

[27] P. Atanassov, *Experimental Assessment of Worst-Case Program Execution Times*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[28] R. P. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Softw. Test, Verif. Reliab*, vol. 9, no. 4, pp. 263–282, 1999.

[29] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

[30] N. Tracey, J. Clark, J. McDermid, and K. Mander, "A search-based automated test-data generation framework for safety-critical systems," in *Systems engineering for business process change: new directions*, (New York, NY, USA), pp. 174–213, Springer-Verlag New York, Inc., 2002.

[31] A. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[32] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[33] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, (Porto Sani, Greece), p. 3, Oct. 2008.

[34] G. Bernat, A. Colin, and S. M. Petters, “pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems,” ycs-2003-353, Department of Computer Science, University of York, Feb. 2003.

[35] J. Kennedy and R. C. Eberhart, “Particle swarm optimization,” in *Proc. of the IEEE Int. Conf. on Neural Networks*, (Piscataway, NJ), pp. 1942–1948, IEEE Service Center, 1995.

[36] M. Clerc and J. Kennedy, “The particle swarm - explosion, stability, and convergence in a multidimensional complex space,” *IEEE Trans. Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[37] Y. Shi and R. C. Eberhart, “Parameter selection in particle swarm optimization,” *Journal Lecture Notes in Computer Science*, vol. 1447, p. 591, 1998.

[38] J. Kennedy and R. C. Eberhart, “A discrete binary version of the particle swarm algorithm,” *Systems, Man, and Cybernetics, 1997.'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, vol. 5, 1997.

[39] R. C. Eberhart and Y. Shi, “Comparing inertia weights and constriction factors in particle swarm optimization,” in *Proc. of the 2000 Congress on Evolutionary Computation*, (Piscataway, NJ), pp. 84–88, IEEE Service Center, 2000.

[40] Y. Fukuyama, S. Takayama, Y. Nakanishi, and H. Yoshida, “A particle swarm optimization for reactive power and voltage control in electric power systems,” in *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99* (W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, eds.), (San Francisco, CA), pp. 1523–1528, Morgan Kaufmann, 1999.

[41] X. Hu, Y. Shi, and R. Eberhart, “Recent advances in particle swarm,” in *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, (Portland, Oregon), pp. 90–97, IEEE Press, 20-23 June 2004.

[42] R. Eberhart and J. Kennedy, “A new optimizer using particle swarm theory,” pp. 39–43, 1995.

[43] K. Veeramachaneni, L. Osadciw, and G. Kamath, “Probabilistically driven particle swarms for optimization of multi valued discrete problems: Design and analysis,” 2009.

[44] J. Kennedy, "Small worlds and mega-minds: effects of neighborhood topology," in *1999 Congress on Evolutionary Computation*, (Piscataway, NJ), pp. 1931–1938, IEEE Service Center, 1999.

[45] T. Krink, J. S. Vesterstrom, and J. Riget, "Particle swarm optimisation with spatial particle extension," in *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*, (Washington, DC, USA), pp. 1474–1479, IEEE Computer Society, 2002.

[46] Wikipedia, "Article on wikipedia about gray code with references to its use in genetic algorithms." `http://en.wikipedia.org/wiki/Gray_code#Genetic_algorithms`, last accessed in May 2010.

[47] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pp. 69–73, 1998.

[48] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," in *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, (New York, NY, USA), pp. 132–140, ACM, 2001.

[49] M. Zolda, S. Bünte, and R. Kirner, "Towards adaptable control flow segmentation for measurement-based execution time analysis," in *Proc. 17th International Conference on Real-Time and Network Systems (RTNS)*, (Paris, France), Oct. 2009.

[50] W. A. Halang and K. M. Sacha, *Real-time systems: implementation of industrial computerised process automation*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1992.

[51] B. Korel, "Automated software test data generation," *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[52] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *GECCO*, pp. 1121–1128, 2007.

[53] M. Zolda, S. Bünte, and R. Kirner, "Context-sensitivity in ipet for measurements-based timing analysis," 2010.

[54] J. Kennedy, "Minds and cultures: Particle swarm implications," *AAAI Technical Report FS97-02*, 1997.

[55] A. Torn and A. Zilinskas, *Global Optimization.* Berlin, Germany: Springer-Verlag, 1989.

[56] J. Wegener and F. Mueller, “A comparison of static analysis and evolutionary testing for the verification of timing constraints,” *Real-Time Syst.*, vol. 21, no. 3, pp. 241–268, 2001.

[57] J. Kennedy, R. C. Eberhart, and Y. Shi, *Swarm Intelligence.* Evolutionary Computation Series, San Francisco: Morgan Kaufman, 2001.