Search-Based Temporal Testing of Multicore Applications

Komsan Srivisut

PhD

University of York

Computer Science

September 2017

Dedication

For my parents, Sutan and Rawadee, and my brother, Komsin.

Abstract

Multicore systems are increasingly common as a modern computing platform. Multicore processors not only offer better performance-to-cost ratios relative to single-core processors but also have significantly minimised space, weight, and power (SWaP) constraints. Unfortunately, they introduce challenges in verification as their shared components are potential channels for interference. The potential for interference increases the possibility of concurrency faults at runtime and consequently increases the difficulty of verifying. In this thesis, search-based techniques are empirically investigated to determine their effectiveness in temporal testing—searching for test inputs that may lead a task running on an embedded multicore to produce extreme (here longest) execution times, which might cause the system to violate its temporal requirements. Overall, the findings suggest that various forms of search-based approaches are effective in generating test inputs exhibiting extreme execution times on the embedded multicore environment. All previous work in temporal testing has evolved test data directly; this is not essential. In this thesis, one novel proposed approach, i.e. the use of search to discover high performing biased random sampling regimes (which we call 'dependent input sampling strategies'), has proved particularly effective. Shifting the target of search from test data itself to strategies proves particularly well motivated for attaining extreme execution times. Finally, we present also preliminary results on the use of so-called 'hyper-heuristics', which can be used to form optimal hybrids of optimisation techniques. An extensive comparison of direct approaches to establishing a baseline is followed by reports of research into indirect approaches and hyper-heuristics. The shift to strategies from direct data can be thought of as a leap in abstraction level for the underlying temporal test data generation problem. The shift to hyper-heuristics aims to boost the level of optimisation technique abstraction. The former is more fully worked out than the latter and has proved a significant success. For the latter only preliminary results are available; as will be seen from this work as the whole computational requirements for research experimentation are significant.

Contents

List of Tables

List of Figures

Acknowledgements

I would first like to sincerely thank my supervisors for their continuous support, excellent guidance and encouragement throughout my studies. I owe a particular debt to my external supervisor, Professor John A Clark, who was my first supervisor before he moved to the University of Sheffield. I am equally indebted to Professor Richard F Paige, who was my prior internal assessor and had stepped in to support me during the tough time in the final year of my PhD. This thesis is as much their effort as it is mine; without their patience and time, I would not be able to complete this thesis.

I would also like to acknowledge my internal and external examiners, Professor Susan Stepney and Dr Leonardo Bottaci, respectively, for their invaluable and insightful comments, which inspired me to broaden my research from various perspectives.

My gratitude also goes to all my colleagues, especially Dr Nathan Burles, who was always willing to help and share his expertise on the P4080 development board and Linux-related solutions. I learnt a lot from him.

Besides, my studies would have been impossible without the financial support of the Ministry of Science and Technology and the Royal Thai Government. As part of the Dynamic Adaptive Automated Software Engineering (DAASE) project, a special thanks must also go to the Engineering and Physical Sciences Research Council (EPSRC) for granting this work.

Finally, I would like to genuinely thank my parents and my brother for their wise counsel and sympathetic ear. Although they are about six/seven hours ahead of me, they are always there for me—through thick and thin.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Contributions from this thesis have been published in the following conference proceedings:

- Komsan Srivisut, John A Clark and Richard F Paige. Search-Based Temporal Testing in an Embedded Multicore Platform. In *Proceedings of the 21st European Conference on the Applications of Evolutionary Computation (EvoApplications 2018)*, pages 794–809, April 2018. [1]
  (Based on research described in Chapter 3 of this thesis.)

- Komsan Srivisut, John A Clark and Richard F Paige. Dependent Input Sampling Strategies: Using Metaheuristics for Generating Parameterised Random Sampling Regimes. In *Proceedings of the Genetic and Evolutionary Computation Conference 2018 (GECCO '18)*, pages 1451–1458, July 2018. [2]
  (Based on research described in Chapter 4 of this thesis.)

Chapter 1

Introduction

This chapter addresses the general problem of verifying the temporal behaviour of applications running on multicore platforms. It then posits the use of search-based techniques, and empirically investigates their applicability to verifying temporal behaviour. To mitigate such verification issue, a research hypothesis is therefore proposed. It is then followed by a brief overview of the rest of the thesis.

1.1 Motivation

In many real-time embedded systems, timing issues are crucially important. Producing outputs too early or too late may be fatal to human life [3], especially in safety-critical systems, such as avionics and automotive systems. The correctness of system function, therefore, depends not only on logical correctness but also on temporal correctness (i.e. the time when the results are produced) [4, 5].

In order to verify the temporal behaviour of real-time systems, timing analysis is typically the principal timing-related safety evidence [6]. In particular, the Worst-Case Execution Time (WCET) of each task is generally sought [7]. Nevertheless, it is infeasible to determine the *exact* WCET. Execution times of a task may vary depending on the input data and different behaviour of the environment [7]. More specifically, technical characteristics, such as multicore architectures [7, 6], as well as the use of parallelism, distribution and fault-tolerant mechanisms [4], lead to non-deterministic behaviour and consequently complicate the exact determination of the WCET [4]. Rather, either an upper bound or conservative estimate of the WCET is used for safety assurance [7, 6].

Techniques for determining the WCET can be broadly categorised into three: 1) static (analysis-based) approaches; 2) dynamic (measurement-based) approaches; and 3) hybrid approaches [6].

Static methods normally analyse all possible control flow paths through the programming language constructs on the underlying Operating System (OS) without knowledge of the input nor executing the actual application [8, 7]. In many cases, however, static WCET analysis does not always produce a usefully tight upper bound on the WCET [6].

Dynamic methods, on the other hand, verify the temporal quality of embedded systems by executing the task or task parts on a given hardware for some set of inputs [7]. The measured times are then taken and the maximal observed execution times are derived [7]. In the literature, e.g. [4, 9], these methods are also called 'temporal testing'. The ultimate goal of temporal testing is to find test inputs which will cause the system to violate performance timing requirements [4]. A number of research studies have also attempted to combine these (measurement-based) approaches with analysis-based approaches; these are normally known as hybrid methods [7, 6].

There are several methods for performing temporal testing of embedded real-time systems, including constrained random-based temporal testing, stress-based temporal testing, search-based temporal testing and mutation-based temporal testing [4]. In this thesis, search-based temporal testing is the main focus.

Search-Based Software Testing (SBST) is an approach that uses a metaheuristic optimising search technique, e.g. a Genetic Algorithm (GA), to automate a testing task [9]. Search-based approaches have been successfully applied across the spectrum of test case design methods; this includes white-box (structural) testing, black-box (functional) testing, grey-box (combination of structural and functional) testing, as well as non-functional testing (such as temporal testing) [10, 9]. Particularly, search-based temporal testing uses an optimisation algorithm to automatically search for test inputs that will produce extreme execution times, e.g. either the longest or the shortest duration [4]. The aim is to discover whether such test inputs can cause the system to violate its temporal requirements [4].

There is an increasing requirement for high-performance processing while maintaining a reasonable power consumption and all at an affordable price. Additionally, there is a trend towards highly integrated architectures, where resources can be shared by different systems/functions in order to reduce the costs. As a result, most of today's processors are multicore processors, not only in the desktop and server but also in the embedded systems

market [11, 12].

The move towards multicore platforms, unfortunately, raises numerous challenges to the real-time system community, since *interference*, which is the unintended interaction between threads on shared resources in multicores [13], prevents execution times from being composable, predictable or even deterministic [12]. These qualities, i.e. execution times' determinism, predictability and composability, are the main ones expected from platforms for time-critical computing and are also dictated by the safety standards, such as DO-178 for avionics systems [12].

At the hardware level, for example, interference can occur through shared hardware components between the cores of a multicore processor. These interference channels comprise caches and memory systems, interconnection structures, shared peripheral controllers, inter-processor and broadcast interrupts, and thermal control and power management infrastructure.

Fuchsen [14] empirically investigated these cross interference channels over multicore platforms. Evidence from the investigation of [14] showed that *cache sharing* and *cache coherency* are essential causes of interference in dual-core processors, i.e. the Intel[1] and AMD[2] processors. Most multicore processor cache architectures have shared caches, varying with the CPU family. Shared caches generally cause the worst case performance loss (in terms of read and write throughput) through the second core by approximately 50 per cent if the data set is significantly larger than the L2 cache.

Furthermore, multicore systems specifically use a cache coherency protocol, such as Modified, Shared and Invalid protocol (MSI), to maintain coherency between caches. False sharing, which is a performance reduction caused by the coherency protocol, can arise if two cores operate on logically independent data but these data are physically stored in a memory region, which ends up in the same cache line. For instance, the read throughput of the Intel processor drops down to 90 per cent if one core is reading while the other core is writing the same data set. For further details on an enumeration of interference issues, the reader is referred to [14, 12].

[1]The Intel Pentium Dual Core E5300 processor has a per-core 32KB + 32KB L1 data and instruction cache and a shared 2MB unified L2 cache.

[2]The AMD Athlon II X2 processor has a per-core 64KB + 64KB L1 data and instruction cache and a per-core 512KB unified L2 cache.

Interference increases the difficulty of verifying the embedded systems, especially with respect to their temporal quality. More specifically, the additional non-functional dependencies introduced by interference might not only cause common-cause failures but also would lead overloads to be unpredictable (which implies additional delays possibly violating the timing constraints of the systems) [12].

As far as we are aware, based on the literatures [10, 15] and Search-Based Software Engineering (SBSE) repository [16], all previous work in search-based temporal testing so far has applied a metaheuristic algorithm, i.e. either GA or Simulated Annealing (SA), to directly search temporal test data. Also, only a little work in this area has emphasised the multicore's interference issue; the experiments were, however, conducted on a simulator [17, 18].

Therefore, this research investigates the application of a variety of search-based techniques for generating sequences of test inputs, which may give rise to extreme behaviour of fundamental mathematical functions running on a non-deterministic environment of a real multicore platform.

1.2 Research Hypothesis

The experimental study in this thesis is based on the following main hypothesis:

Hypothesis: *Search-based approaches are effective ways of finding test data that attains an extreme execution time for an application's task running on a multicore platform.*

Regarding this research hypothesis, the central objective of this thesis is:

Objective: *To explore the effectiveness of search-based approaches to finding test data that attains an extreme execution time for an application's task running on a multicore platform.*

There are different ways search-based approaches can be brought to bear on our execution time problem. In this thesis, as illustrated in Figure 1.1, we classify search-based approaches into primary three different methods: 1) direct approaches; 2) indirect approaches; and 3) hyper-heuristics. Correspondingly, the study is divided into following separate technical chapters:

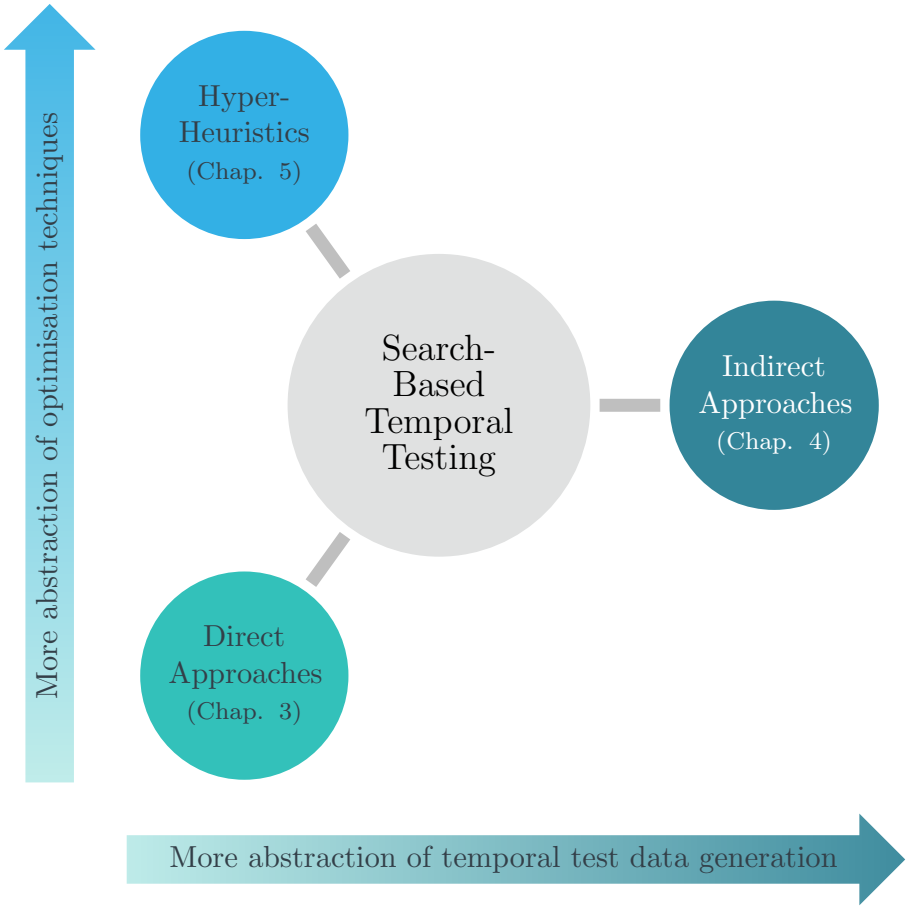


Figure 1.1: Overview of search-based approaches for temporal testing

- Direct approaches (Chapter 3), where the search space is the set of possible *test input vectors* of the Software Under Test (SUT);
- Indirect approaches (Chapter 4), where the search space is a *parametrised family* of test input generation strategies (we seek a *distribution* from which test inputs can be sampled); and
- Hyper-heuristic approaches (Chapter 5), where searches are made over the space of search heuristics (i.e. seeking an effective hybrid technique).

More specifically, as demonstrated in Figure 1.1, in the perspective of optimisation techniques (vertical axis), shifting the target of the search from test data itself (by means of direct approaches) to either strategies (by means of indirect approaches) or heuristics (by means of hyper-heuristics) aims to boost the abstraction level for the search space.

Furthermore, in the perspective of underlying temporal test data generation problem (horizontal axis), indirect approaches can also be thought of as a leap in the level of abstraction in the sense that the selected strategy is used to generate test data which are dependent to each other.

The research objective is accordingly broken down into three sub-objectives, as follows:

Sub-Objective 1: *To explore the effectiveness of applying metaheuristics on temporal testing of a task running on a multicore platform.*

Sub-Objective 2: *To propose a novel approach, named 'dependent input sampling strategy', and to explore its effectiveness on temporal testing of a task running on a multicore platform.*

Sub-Objective 3: *To explore the effectiveness of applying (selective perturbative) hyper-heuristics on temporal testing of a task running on a multicore platform.*

1.3 Brief Overview of the Thesis Chapters

The remaining chapters of this thesis are:

Chapter 2 Literature Review This chapter provides detailed background related to this research, including multicore processors, software testing, SBST and related work.

Chapter 3 Direct Optimisation This chapter aims to assess the effectiveness of applying metaheuristic techniques, i.e. Stochastic Hill Climbing (HC), Steepest Ascent Hill Climbing (SHC), SA and GA, on temporal testing of a task running on a multicore platform.

Chapter 4 Indirect Optimisation This chapter proposes a novel approach, i.e. the application of metaheuristic search to find a dependent input sampling strategy for test data generation with the aim of attaining extreme execution times. We adopt sequential sampling of a task's parameters and allow the sampling of a parameter to depend on the sampling of previous parameters. Using search to find the best subdomains to sample and to determine optimal dependencies forms the focus of this chapter's research. Again, the execution takes place on a multicore platform.

Chapter 5 Hyper-Heuristics The goal of this chapter is to investigate the ability of a hyper-heuristic to perform temporal testing of a task running on a multicore platform.

Chapter 6 Conclusions This chapter concludes the thesis. The contributions made, limitations of the work and future research directions are presented.

Chapter 2

Literature Review

This chapter describes the basic background necessary for understanding the rest of this thesis. The chapter begins with an overview of multicore technologies. After that, there is a description of verification challenges arising with multicore platforms. Next, a short introduction to software testing, together with an overview of the diverse categories of testing, is given. This is followed by a brief description of optimisation and search techniques generally used for SBST, as well as the fundamental concept of hyper-heuristics. Finally, a literature survey of SBST for temporal behaviour is summarised.

2.1 Multicore Processors

Due to the myriad demands for efficient computing performance, especially in the areas of science and engineering (e.g. to model or simulate difficult problems), major chip manufacturers have continually improved their products' performance. For almost a half-century, Gordon Moore, the Intel's co-founder, predicted that the number of transistors per square inch on integrated circuits would grow exponentially; it is popularly known as Moore's Law [19]. Accordingly, advances in silicon processing technology (i.e. technology scaling) allows chip components to be manufactured with tinier size [20], resulting in higher transistor density and shorter distance between components embedded on a chip. This smaller process technology, therefore, enables higher clock frequencies. The technology also allows a more sophisticated design on the chip. For instance, the cache size and number of cache levels could be increasingly integrated, and the number of instruction pipelines[1]

[1]Instruction pipelining is a technique that implements a form of parallelism called 'Instruction-Level Parallelism (ILP)' within a single processor.

could be added into the chip for improving the overall performance of integrated circuits [14].

The trend had largely followed Moore's Law until the last few years when power consumption rose at a faster rate than the clock speeds; it reached its limit. In particular, increasing CPU frequency causes disproportionate power consumption and thermal dissipation loss [14]. In some cases, the power problem could be aggravated by the designs that attempt to dynamically extract extra performance from the instruction stream [21]. Furthermore, existing parallelism and dependencies on code level prevented further performance improvement through parallel execution on instruction level within a single processor [14]. Power hungriness leads in turn to a thermal problem, which requires additional costly infrastructure to effect appropriate heat dissipation. Fred Pollack, a lead engineer and fellow at Intel, empirically proved that the performance is roughly proportional to the square root of the increase in complexity of the micro-architectural design; this is generally known as Pollack's Rule [22].

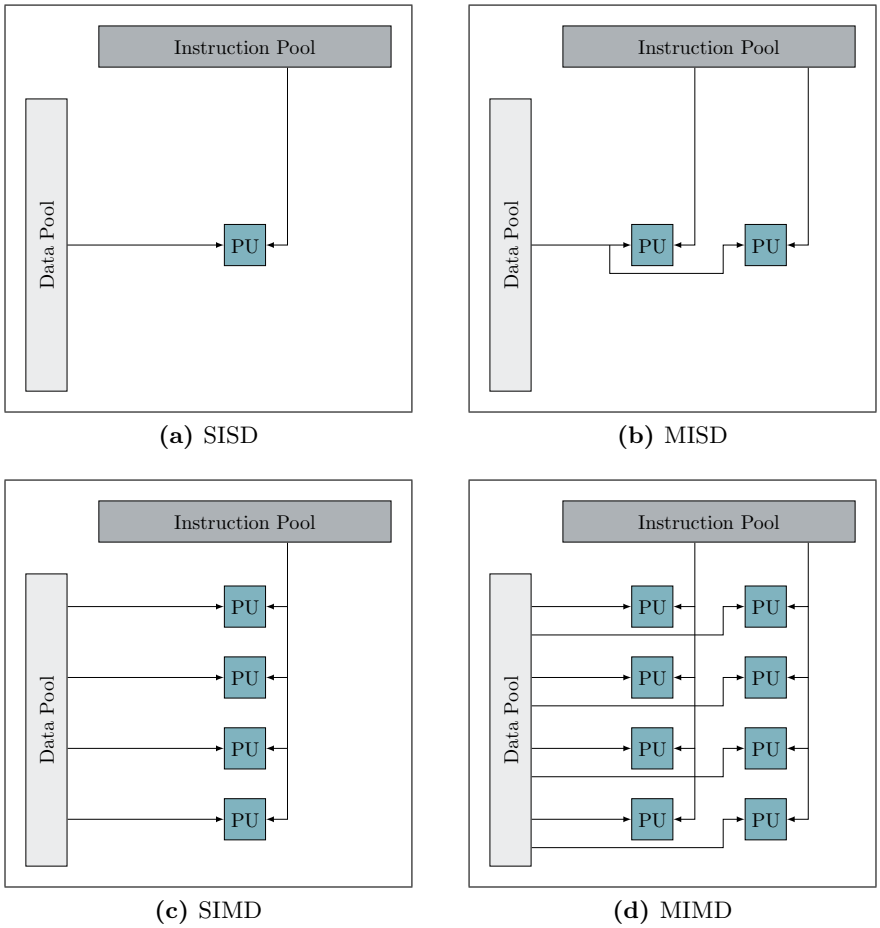
Seeking to improve performance by simply increasing frequency through technology scaling is insufficient and is not a viable solution to the problems faced today. The trend has led to a new approach in exploiting fabrication processes, where the area cost reduction obtained from scaling is used to increase the number of cores, to provide more processing bandwidth [20]. Consequently, multicore processors have been introduced to address the issues described above. By adding more processing units on a single die to provide a higher level of parallelism, multicore systems are capable of scaling performance while reducing Space, Weight and Power (SWaP) consumption. Multicore has, therefore, become the typical design choice for a broad range of computing domains, such as high-end servers, desktops, mobiles, and embedded systems. Many believe that single-processor systems are obsolete [20, 23].

2.1.1 Architectures for Parallelism

For the past several decades, parallel computation has, in fact, been exploited in order to improve computing performance. Parallel computing is the simultaneous use of multiple computing resources to solve a computational problem. The computing resources typically range from a single computer with multiple processors/cores (as a stand-alone computer) to an arbitrary number of such computers connected by a network (as a cluster). The classifications and forms of parallelisation are briefly summarised below.

Computer Architecture Classifications There are different ways to classify parallel computer architectures, for example, based on the instruction and data streams, the structure of computers, how memory is accessed and grain size [24]. One of the more widely used classifications is Flynn's Taxonomy [25]. Based on the notion of a stream of information, there are two types of information flow into a processor of a computer: instructions and data. Flynn's Taxonomy classifies computing machines into four categories according to how instructions and data are processed at a single point in time, as illustrated in Figure 2.1. (Note: PU stands for a processing unit.)

Single Instruction, Single Data Traditional uniprocessor machines, such as old PCs and old mainframes, are in this Single Instruction, Single Data (SISD) class [26]. There is no parallelism in these traditional sequential computers since the CPU can process only one data stream during a given clock cycle [26].



(a) SISD **(b)** MISD

(c) SIMD **(d)** MIMD

Figure 2.1: Flynn's taxonomy (adapted from [25])

Multiple Instructions, Single Data This class is an unusual architecture because multiple instruction streams commonly need multiple data streams to be useful [26]. Therefore, Multiple Instructions, Single Data (MISD) is generally used more as a theoretical model than a practical computing platform [26].

Single Instruction, Multiple Data As a single instruction stream has the ability to process multiple data streams simultaneously in a single clock, the Single Instruction, Multiple Data (SIMD) machines are best suited for specialised problems characterised by a high degree of regularity, such as signal, image and video processing [26].

Multiple Instructions, Multiple Data The most common parallel computing platform is Multiple Instructions, Multiple Data (MIMD) architecture, where multiple processors execute different instructions on different data simultaneously [26]. Examples of this category include most current supercomputers, networked parallel computer clusters and grid, multi-processor SMP computers and multicore PCs [26].

Forms of Parallelism According to Flynn's taxonomy, the processor resources used in the latter two categories (i.e. SIMD and MIMD machines) are capable of supporting parallel computation [26]. In order to make the most efficient use of these processor resources, ILP is typically employed to realise a higher performance on single-core processors, and Thread-Level Parallelism (TLP) is afterwards introduced to reinforce the performance on multicore processors.

Regarding the ILP, multiple instructions from a single program are executed by the processor in a single clock cycle. Superscalar, Very Long Instruction Word (VLIW) and out-of-order execution are examples of micro-architectural techniques (which will be presented in Section 2.1.2) that are used to exploit the ILP [27]. The TLP is, by the way, a parallelism on a coarser scale since a sequence of programmed instructions that can be managed independently by an OS scheduler is generally defined as a thread of execution [27]. Therefore, different threads are executed in parallel on different processors according to the TLP. The threads could be either from separated single-threaded applications or from the same multi-threaded applications [27].

2.1.2 Multicore Architecture

In recent years, a wide range of multicore architectures has been delivered to commercial markets, e.g. embedded systems, general-purpose desktops and servers. With diverse demands, there are numerous ways to implement multicore processors for the target market segments. Each realm has its specific requirements.

For example, power is not an overriding concern for general purpose computing, such as desktop and server multicores, while it is a vital issue for mobile and embedded applications because many are intended to run from batteries [21]. In high-performance computing, which is more specialised, a significant number of cores and very high power are required to deliver the highest performance [21].

Figure 2.2 illustrates an example of Multiprocessor System-on-Chip (MPSoC) architecture, i.e. the QorIQ P4080 multicore processor, which is the primary multicore platform used for conducting empirical experiments in this thesis.

Multicore architectures can be classified in a number of ways. There are generally five major distinguishing attributes of multicore architectures: application class; power/performance; processing elements; memory system; and accelerators/integrated peripherals [21].

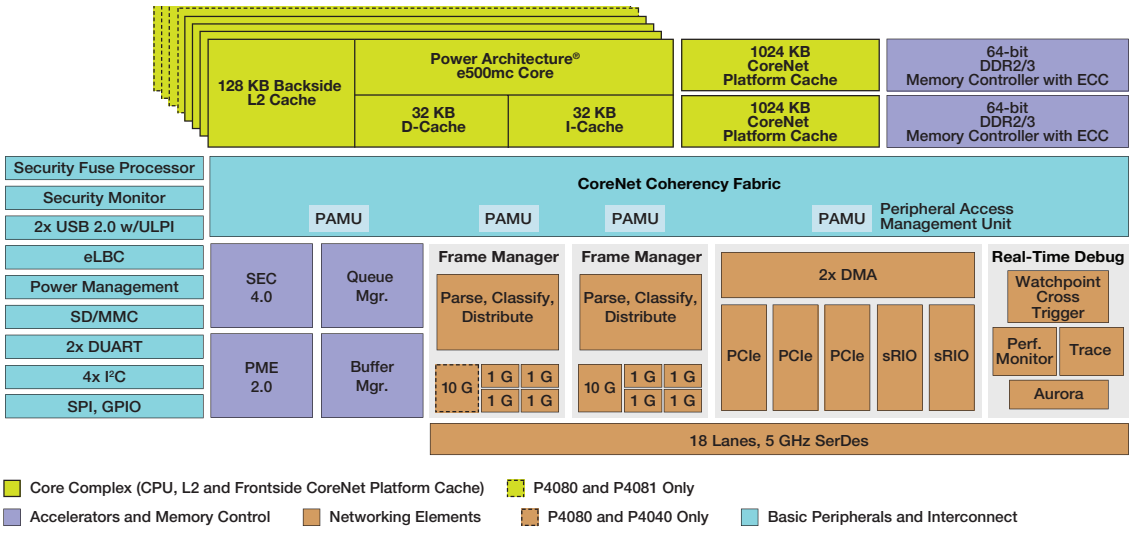


Figure 2.2: Block diagram of QorIQ P4080 (reprinted from [28])

Application Class One of the essential characteristics that can be used to differentiate between multicore architectures is their envisaged application domain. There are broadly two divisions of applications for which the processor is targeted, i.e. data processing dominated and control processing dominated applications [21].

In most cases, although no application can purely fit into these classes, it is still valuable to think of applications as falling into the divisions in order to understand how different multicores design aspects can affect performance [21]. In a more fine-grained level, execution phases of an application may genuinely fit into these neat categories [21]. For example, the H.264/AVC[2] video codec [29], which is one of the execution phases for a wide variety of applications, such as high definition systems (e.g. HDTV, Blu-ray Disc players and HD DVD players), low resolution portable devices (e.g. Sony's PSP and Apple's iPod), streaming internet services (e.g. Vimeo and YouTube), web software (e.g. Adobe Flash Player and Microsoft Silverlight) and HDTV broadcasts (over terrestrial, cable and satellite), is data dominated [21].

Typically, the computation of the data processing dominated application category is a sequence of operations on a data stream with little or no data reuse [21]. In order to handle the large amounts of data, the operations can be performed in parallel and require high throughput and performance [21]. Therefore, this kind of application favours having as many processing elements as practical with regards to the desired power/performance ratio [21]. Examples of this group are graphics rasterisation, image processing, audio processing and wireless baseband processing [21].

Conversely, the algorithms for the control processing dominated application type needs to keep track of large numbers of states together with conditional branches and often have a high amount of data reuse (which complicates the parallelism) [21]. Therefore, this type of application favours a smaller number of general-purpose processing elements to handle the unstructured nature of control dominated code [21]. Example applications include file compression/decompression, network processing and transactional query processing [21].

Power/Performance Performance and power are strict requirements for many applications and devices [21]. For instance, a smartphone has a limited battery budget but must demonstrate good performance for numerous features and services [21].

[2]H.264 (also known as MPEG-4 AVC) is a video compression standard that is currently one of the most commonly used formats for the recording, compression and distribution of video content.

Processing Elements At a fundamental level, a processing element (or microprocessor) is the physical implementation of instructions, which specify what exactly tasks the microprocessor is allowed to perform [21]. The set of all instructions is called the 'instruction set' [21]. The Instruction Set Architecture (ISA) is defined by the combination of the instruction set and other information, including the native data types, registers, addressing modes, memory architecture, interrupt and exception handling and external I/O, and defines the boundary between software and hardware. The micro-architecture is one way in which a given ISA is implemented on a processor [30].

Architecture The instruction sets can be classified into five major types according to design philosophies, including a legacy ISA, custom defined ISA, Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC) and special instructions [21]. These architectural design parameters are summarised in Table 2.1.

Table 2.1: Summary of pros and cons of various multicore architectural design parameters (adapted from [21])

ISA	Pros	Cons
Legacy	Compiler and software support.	Maybe inefficient for targeted applications requiring high performance.
Custom	Can be highly optimised for target applications.	Compiler and software support can be non-existent.
RISC	Easier micro-architectural design; and easier compiler design.	Code size can be large; and inefficient for certain applications.
CISC	More instructions that may allow for better optimisation; and smaller code size.	Complex micro-architectural design to support all instructions; and compiler design complicated.
Special Instruction	Allows highly optimised core for targeted functions.	Complex to design; and may require hand coding due to limited/no compiler support.

Typically, the ISA of conventional multicore processors is a legacy ISA [21]. The legacy ISAs are modified a bit from the corresponding uniprocessor in order to support parallelism, e.g. by adding atomic instructions for synchronisation [21]. The existent implementations and available programming tools of the uniprocessor system are, therefore, by-products of the legacy ISAs [21]. In addition, an ISA may be custom defined for specific applications, but it also requires particular compiler and software support [21].

ISA can also be divided into CISC and RISC designs. The former embraces creating an instruction set in which individual instructions are very powerful but quite complex [21]. Oppositely, the latter creates instruction sets in which the individual instructions are minimal; their simplicity, however, allows many optimisations to be performed [21]. Although CISC and RISC accomplish the same amount of work, there are some trade-offs between the approaches. For instance, RISC probably gives rise to programs with larger code sizes owing to the need to emulate more complex instructions with the smaller set of RISC instructions. RISC instructions, however, most likely execute more quickly than CISC since they provide an easier target for compilers and allow for easier micro-architectural design [21].

In order to improve performance for common operations, vendors have continually added extensions to the base-defined ISA [21]. These extra instructions allow for better performance/power consumption ratio [21]. However, using these extensions to maximum effect may require hand coding due to limited or no compiler support [21].

Micro-Architecture In general, a given ISA may be implemented with different micro-architectures, depending on the different goals of a given design [21]. The traditional goals like performance and power are thus mainly governed by the micro-architecture [21]. Furthermore, the micro-architecture of each processing element can be tailored to support its targeted application domain [21].

There are a variety of types of micro-architectural processing elements. Many of their designs typically utilise an instruction pipeline; these include in-order, out-of-order and VLIW processing elements. By performing multiple operations at the same times without reducing instruction latency—the time to complete a single instruction from start to finish, the pipeline technique is able to increase instruction throughput—the number of instructions that can be executed in a unit of time [31]. In addition, pipelining can be modified to get the more desirable performance by two main parameters. [21].

First, performance can be improved by adding multiple pipelines [21]. A superscalar execution[3] is then created since more than one instruction is allowed to fetch, as well as to issue simultaneously [21]. However, extra logic is necessarily required to assure correct code execution in the pipelines [21]. Second, increasing the number of pipeline stages can also improve the performance as the logic per stage is reduced [21]. Additionally, a clock is able to be faster if the instruction sequence is broken by branches [21].

The in-order processing element is the simplest type, which instructions are decoded and executed in program order [21]. This processing element type requires a small die area and low power [21]. On the contrary, the out-of-order processing element insists on a substantial die area and power-hungry circuitry owing to the dynamic scheduling [21]. In particular, multiple instructions are dynamically found and scheduled in an out-of-order manner to keep the pipelines full [21].

In order to avoid the complexity of the extra logic introduced by the dynamic scheduling needed to properly execute the instruction streams while still providing increased performance over superscalar architectures, SIMD or VLIW can be applied [21]. The SIMD architecture, as aforementioned, technically processes multiple data points with one instruction in lanes defined by partitions of very wide registers [21]. This architecture is well suited for data-intensive applications, but highly not for general-purpose processing [21]. The VLIW processors, on the other hand, support the MIMD architectures by using multiple pipelines without having the forwarding, scheduling and hazard detection logic of a typical superscalar core [21]. Therefore, multiple data points can be processed with several instructions simultaneously [21]. The complexity, however, has been moved to the compiler since it determines what operation each functional unit performs in each instruction cycle [21]. As a result, underutilised problems can be incurred if the compiler cannot find sufficient parallelism [21]. Micro-architectural design parameters are summarised in Table 2.2.

Multicore processors come in two flavours, namely homogeneous and heterogeneous architectures [21]. In a homogeneous architecture, a number of identical cores are utilised in order to obtain a power advantage without loss of performance [21]. The identity of the cores means each processor has the same instruction set and data structures [21].

[3]The keys to superscalar execution are: 1) an instruction fetching unit which can fetch more than one instruction at a time from the cache; 2) an instruction decoding logic which can decide when instructions are independent and thus executed simultaneously; and 3) sufficient execution units to be able to process several instructions at one time [32].

Table 2.2: Summary of pros and cons of various multicore micro-architectural design parameters (adapted from [21])

Micro-architecture	Pros	Cons
In-order	Low to medium complexity; low power; and low area (so many can be placed on the die).	Low to medium single thread performance in general.
Out-of-order	Very fast single thread performance from dynamic scheduling of instructions.	High design complexity; large area; and high power.
SIMD	Very efficient for highly data-parallel/vector code.	Can be underutilised if the code cannot be vectorised; and not applicable to control-dominated applications.
VLIW	Can issue many more instructions than out-of-order due to reduced complexity.	Required advanced compiler support; and may have worse performance than narrower out-of-order core if a compiler cannot statically find ILP.

On the other hand, cores in a heterogeneous architecture are different, i.e. they may have distinct instruction sets and data structures, to satisfy the high-performance and low-power requirements [21]. For example, a processor may have a common CPU core for applications, a Digital Signal Processing (DSP) core for maths, a graphics core for graphics work and an I/O processor core to handle specific I/O interfaces. However, the programming model for heterogeneous multicore is much more complicated [21].

Memory System Cache memory plays an important role as it is an intermediary between main memory and processing elements [32]. The idea is to have a local place for storing items that will be used in the near future [32]. This brings about reducing the number of accesses to the main memory, which is relatively slow owing to the memory type it is. Examples of main memory include extended data-out (EDO), synchronous dynamic RAM (SD-RAM), double data rate SD-RAM (DDR-SDRAM) and Rambus dynamic RAM (RD-RAM) [32]. The memory latency is, therefore, lessened and subsequently, the response time is improved [32].

Multicore processors use different levels of local memories to decrease the average memory access latency [32]. These local memories can be considered as a hierarchical system, where the CPU core is at the top of the hierarchy, followed by one or more levels of cache memories, and the main memory (or physical memory) is at the bottom [32], as depicted in Figure 2.3.

The number of cache levels depends on the distance in cycles between the main memory and each processing element [32]. The longer the distance in cycles, the higher the need for more cache levels [32]. Cache memories can be shared between CPUs or be local to a CPU.

Generally, the level 1 (L1) cache is directly connected to the CPU and is usually small, fast and dedicated to specific cores [21]. Subsequent levels—the level 2 (L2) cache and so on—can be dedicated or shared among the cores [21]. These higher cache levels are normally larger but slower [21].

The data transferred between the main memory and these different cache levels are in fixed-size chunks, named 'cache lines' [14]. A cache entry, which includes the copied data and the requested memory location, will be created when a cache line is copied from the memory into the cache [14].

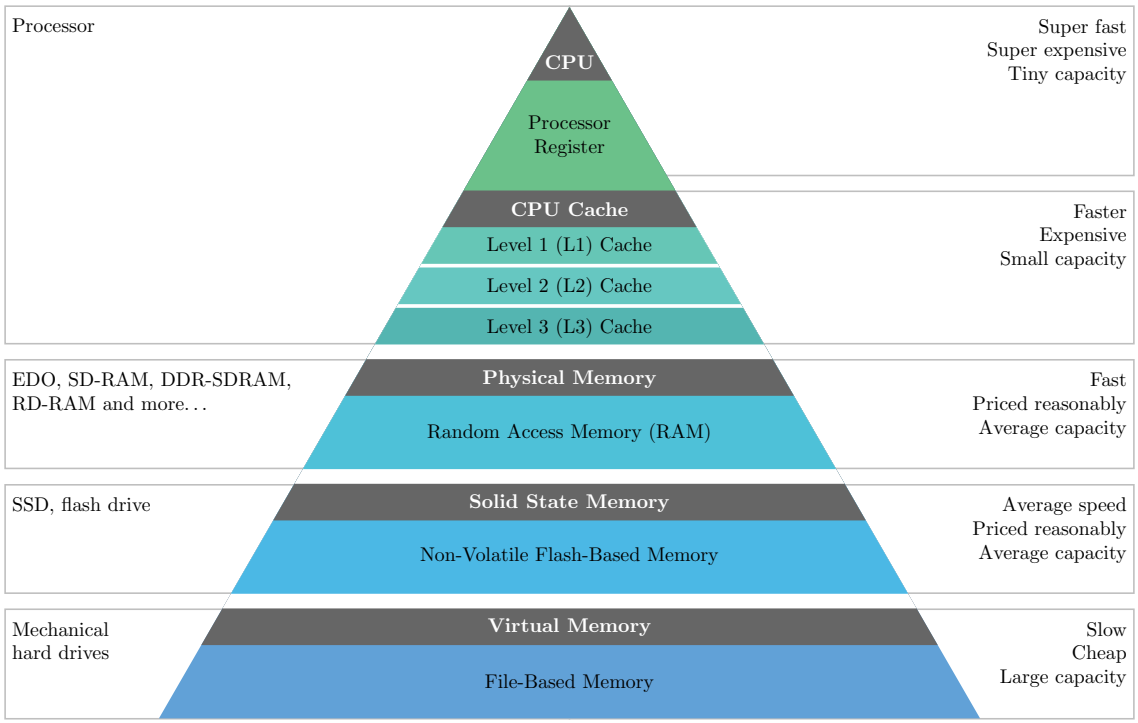


Figure 2.3: Memory hierarchy (adapted from [33])

Whenever the processor needs to read or write a location in main memory, a corresponding entry in the cache is first checked to determine whether the memory location is in the cache [14]. If the memory location is found (called a 'cache hit'), the processor immediately reads or writes the data in the cache line [14]. When the data request on a given level of the cache cannot be satisfied, the request is delegated to the next level instead; otherwise, the cache will allocate a new entry and copy in data from main memory (generally called a 'cache miss') [14]. The request is then fulfilled from the contents of the cache [14].

In addition, the cache memory system is linked with the processing elements and other shared resources through the interconnection. There are diverse topologies for the general communications among these components via the intra-chip interconnect, such as bus, crossbar, ring and Network-on-Chip (NoC) [21]. In terms of simplicity and performance, each form of topologies has its particular benefits and drawbacks, as summarised in Table 2.3. The bus, for instance, is the simplest to design but is not suitable for multicore with a large number of processing elements because of the limitations in bandwidth and latency [21]. On the other hand, NoC is more suitable for a large number of processing elements but is more challenging to design [21].

Table 2.3: Summary of pros and cons of interconnect design decisions (adapted from [21])

Interconnect	**Pros**	**Cons**
Bus	Easy to implement; and all processor see uniform latencies to each other and attached memories.	Low bisection bandwidth; and supports a small number of cores.
Ring	Higher bisection bandwidth than the bus; and supports a large number of processors.	Non-uniform access latencies; high variance in access latencies; and requires routing logic.
NoC	High bisection bandwidth; supports a large number of cores; and non-uniform latencies are lower variance than the ring.	Requires sophisticated routing and arbitration logic.
Crossbar	Highest bisection bandwidth; can support a large number of cores; and uniform access latencies.	Requires sophisticated arbitration logic; and needs a large amount of die area.

Accelerators/Integrated Peripherals For highly specialised processors or Application-Specific Integrated Circuits (ASICs), accelerators or integrated peripherals are typically required to provide extra performance that software cannot emulate efficiently [21]. These components usually require very low power consumption but have a massive impact on overall performance in many cases [21]. Graphics rasterisers, codec accelerators and memory controllers are examples of these components [21].

2.1.3 Multiprocessing Architectures

In OS level, multiprocessing approaches have been applied to multicore systems in order to get the most performance possible out of the hardware for parallel computing [34]. There are three fundamental models for multiprocessing, which can be mapped to the multicore: Asymmetric Multiprocessing (AMP), Symmetric Multiprocessing (SMP) and Bound Multiprocessing (BMP) [34].

Asymmetric Multiprocessing In an AMP model, each core is treated as a separate processing element [34]. Multiple instances of the same application (i.e. the same type and version of OS) are thereby enabled to be replicated across multiple cores and to operate on separate sets of data; this is sometimes referred to as homogeneous AMP [34]. In addition, there is also heterogeneous AMP, where non-identical OSs (i.e. either different OSs or different versions of the same OS) can be run on different cores [34]. Since the cores operate independently, the software development effort, which is required to port to a multicore architecture, is minimised at the application level [34].

However, hardware resources, such as L2 cache, memory buses and certain peripherals, are required to share among individual cores in multicore AMP systems [34]. Particularly, in order to ensure that access and control of the shared resources are handled correctly, software must be designed and implemented carefully; otherwise, an incorrect software operation could bring about the propagation of a fault from one core to another [34]. Furthermore, an appropriate Inter-Processor Communication (IPC) mechanism is also required to utilise an underlying hardware IPC transport for communication among applications running on different cores [34]. Figure 2.4 shows an example of the AMP multicore system.

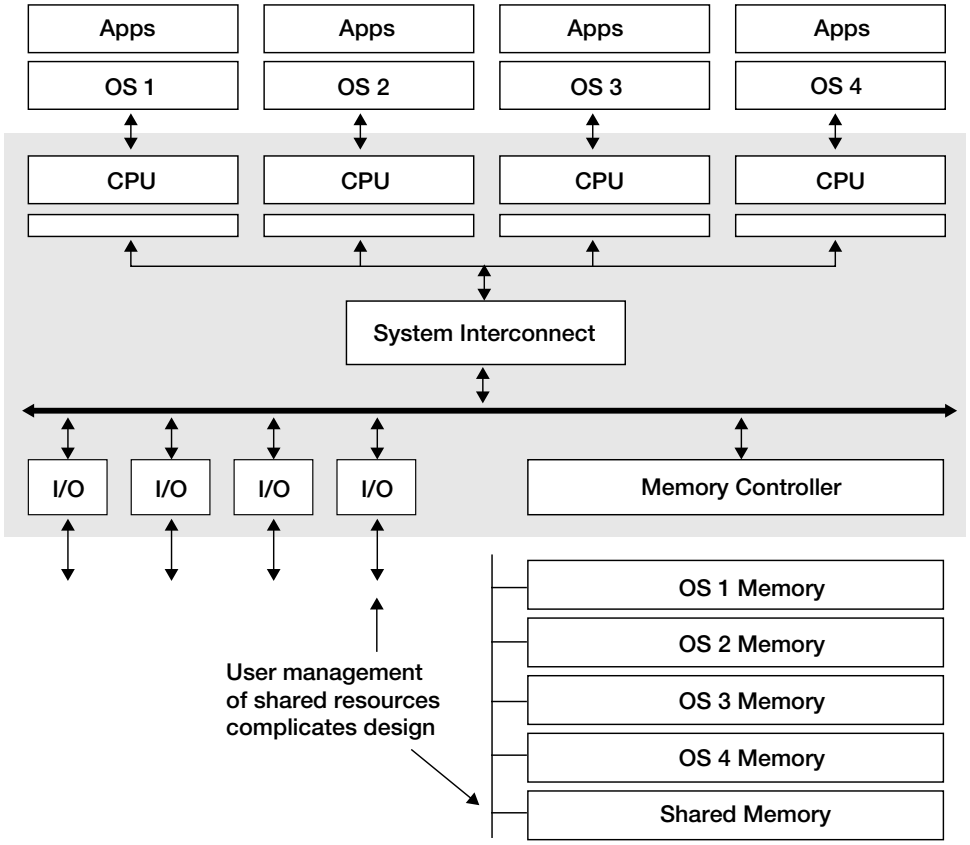


Figure 2.4: AMP (reprinted from [34])

Symmetric Multiprocessing An SMP model, on the other hand, involves the use of a single instance of OS running across multiple processor cores [34]. Hence, the SMP model provides a homogeneous environment for applications [34]. Applications processes and threads are also scheduled to execute across these multiple cores by OS kernel [34]. Rather than being restricted to a single CPU core, the SMP-capable OS will schedule the highest-priority ready thread to execute on the first available core [34]. In addition, since the OS has insight into all system elements at all times, the SMP model is able to dynamically allocate resources to specific applications rather than to CPU cores [34]. These enable greater utilisation of multicore systems [34]. Also, since there is no longer a need for a heavy networking protocol between applications running on different cores, the memory footprint is reduced, all inter-core IPCs are local, and as a result, the performance is dramatically improved [34]. However, a significant software development effort is unfortunately required to migrate single-core architecture applications to a multicore SMP architecture effectively due to the lack of parallelism support in the original software design. Figure 2.5 shows an example of the SMP multicore system.

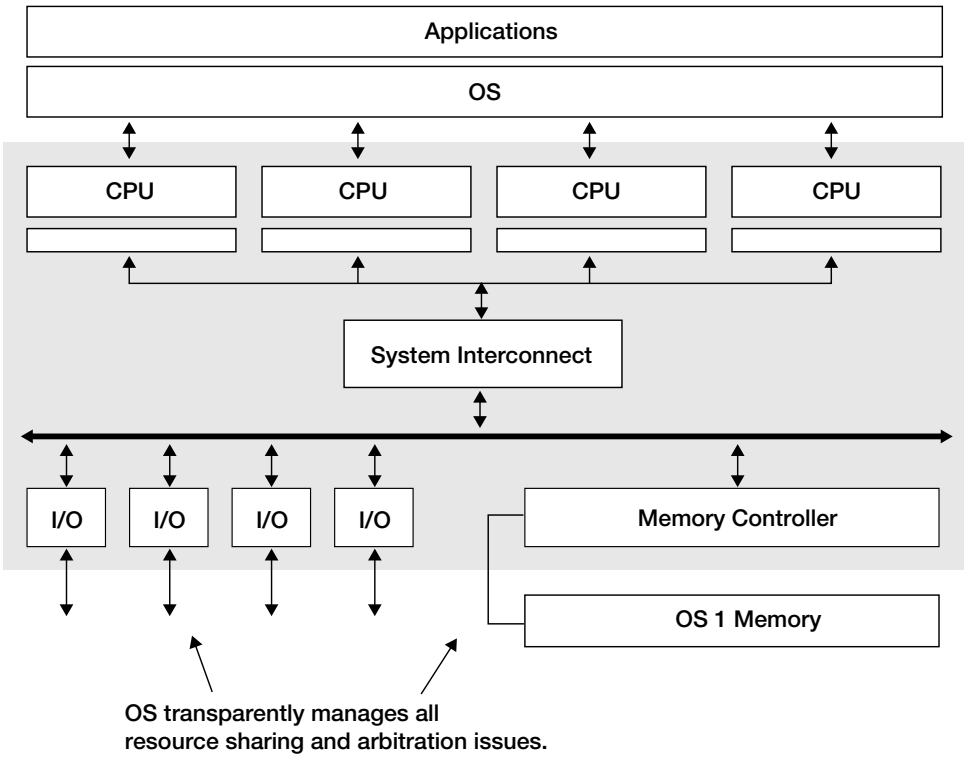


Figure 2.5: SMP (reprinted from [34])

Bound Multiprocessing A BMP model takes full advantages of parallelism of multicore hardware by offering the benefits of SMP, where the resource management is transparent, and of AMP, where designers have the ability to lock any application (and all of its threads) to run in a specific core [34]. In particular, the BMP model allows some threads to migrate from one processor to another, while other threads are restricted to one or more processors [34].

The choice of AMP, SMP and BMP should depend on the problem at hand [35]. For example, AMP works well with legacy applications but has limited scalability beyond two CPUs, whereas SMP offers transparent resource management but any software that has not been properly designed for concurrency might have problems [35]. BMP offers many of the same benefits as SMP, but guarantee that uniprocessor applications will behave correctly; greatly simplifying the migration of legacy software [35]. Table 2.4 illustrates the differences among these multiprocessing models for striking the optimal balance between performance, scalability and ease of migration.

Table 2.4: Summary of multiprocessing features (adapted from [35])

Feature	AMP	SMP	BMP
Seamless resource sharing	-	Yes	Yes
Scalable beyond dual CPU	Limited	Yes	Yes
Legacy application operation	Yes	In most cases	Yes
Mixed OS environment	Yes	-	-
Dedicated processor by function	Yes	-	Yes
Inter core messaging	-	Yes	Yes
Thread synchronisation between CPUs	-	Yes	Yes
Load balancing	-	Yes	Yes
System-wide debugging and optimisation	-	Yes	Yes

The QorIQ P4080 multicore processor, which is a multicore platform used in this research, can be flexibly configured to meet many system application needs [28]. However, only the SMP model is mainly focused since the aim of this research is to observe the interference occurred in a system that shares memory and other resources. Hence, the SMP Linux is set up to be the main Real-Time Operating System (RTOS) for the P4080 systems as it is a general purpose system with a broad application base and runs on a very large number of hardware platforms [36].

Note: there is another OS option for a more specific purpose of the P4080 processor, i.e. Enea's OSE for Multicore [36]. The Enea OSE is a message-passing based real-time system optimised for data plane processing with tougher requirements on stability, determinism and low kernel overhead. [36]. Processing of user data packets and control signalling within both telecom and datacom areas are examples of high-performance data plane applications [36].

2.1.4 Verification Challenges

Although multicore architectures undertake to deliver high computing performance through a combination of parallelisms, i.e. ILP and TLP, there is a potential for interference that increases the difficulty of verifying and certifying high-integrity systems. The interference and its challenges are summarised in brief below.

Interference Interference is an unintended interaction between threads and can arise from the variety of channels in multicore architectures [13]. The potential channels for interference include:

Hardware Interference Channels At the hardware level, interference can occur through shared hardware components between the cores of a multicore processor [14]. These interference channels include memory caches, interconnection structures, shared peripheral controllers, inter-processor interrupt, broadcast interrupts, thermal control system and power management infrastructure [14, 13].

For memory caches, there are two main cache interference channels: shared caches and cache coherency [14]. First, the cache sharing interference may occur either constructive or destructive ways [13]. Constructive interference occurs when one processor loads data (i.e. cache line) into a shared cache, which is later used by other processors sharing the same cache; hence, cache misses are avoided [14]. On the other hand, destructive interference can increase the number of cache misses since there are cache conflicts from competing to use the shared cache for the own cache line (i.e. cache entry) among processors [14].

Second, data inconsistency usually occurs if the value of a cache line in a private cache (L1) on one core is changed and the same cache line is also present in the private caches of the other cores [14]. Although the use of cache coherency (e.g. snoopy protocols [37]) is to handle with the data inconsistency problem, the overhead of ensuring cache coherency can reduce performance and thread interference can also be introduced because of the need for cache coherency from the other thread to access the memory [14].

Moreover, congestion and contention can occur through interconnection structures and peripheral controllers because these components are shared between threads running in parallel [13]. Interrupt controllers, which may be used by the OS for communication between threads, as well as mechanisms of thermal control and power management, are also the potential sources of interference [13].

Software Interference Channels Shared software components can also create some interference [14]. For instance, concurrent access of shared data structures, e.g. scheduling data and code in SMP, is a channel for interference at the OS level [13]. Furthermore, at the application level, interference occurs in the form of race conditions, as well as concurrency issues, such as deadlock arising from mechanisms designed to avoid race conditions [13].

Verification and Certification Issues Many critical systems must undergo a stringent evaluation and acceptance process before being deployed. This process is generally referred to as *certification*. Governmental agencies, e.g. the UK Ministry of Defence, or domain-specific bodies, e.g. the Nuclear Regulatory Authority or Civil Aviation Authority, set policies and standards for system acceptability. As far as multicore is concerned, the complexity of confidently verifying systems using such processors is daunting. There has been little published on this subject (compared to other aspects of verification and certification) [13].

The matter is not just technical. As pointed out in [13], there is a good deal of Intellectual Property Right (IPR) in leading-edge multicore design. For many multicore processors, the safety critical market is actually quite small and not particularly influential. Getting access to such information may prove difficult. The reader is referred to [13, 38] for an enumeration of issues.

2.2 Software Testing

Software should perform its intended functions correctly; otherwise, its failures can cause frustration, loss of resources and even loss of life (in safety-critical systems) [39]. Software testing is therefore necessarily required to verify quality and reliability of software. In particular, the testing can prevent (or minimise) the chances of software failures [39].

There are several activities in software testing, such as generating test cases, executing programs with the generated test cases and evaluating the results [39]. Among the range of testing activities, test case generation is considered as one of the most intellectually demanding tasks and also one of the most critical challenges since it can have a strong impact on the effectiveness and efficiency of the whole testing process [40].

According to the IEEE Standard for Software and System Test Documentation[4] (also known as the IEEE Std 829-2008), the term 'test case' is defined as a set of test inputs, execution conditions and expected results developed for a particular objective, such as to exercise a particular program path and to verify compliance with a specific requirement [41]. A set of test cases is simply called a 'test set'[39].

[4]It is an IEEE standard that specifies the form of a set of documents for use in eight defined stages of software testing and system testing.

Test case generation is normally referred to as a process of creating or identifying test data which can satisfy a given test criterion of the SUT [42]; in many cases, the process can be synonymously called 'test data generation'.

In practice, however, *exhaustive testing* (or, in other terms, *complete testing* and *full coverage*) cannot be performed, as it is usually computationally too expensive, and also infeasible due to the fact that the number of potential inputs for most programs is so large and cannot be explicitly enumerated [43]. Hence, *coverage criteria* are ordinarily used to decide which test inputs to be used [43]. Effective use of coverage criteria is believed to make it more likely that test engineers will find faults in a program, and provides informal assurance that the software is of high quality and reliability [39].

Moreover, software testing is expensive and labour intensive; often up to 50 per cent of total software development costs are accounted for the testing phase [40] and even more for safety-critical applications [44]. Automation of test activities as much as possible is thereby desirable because automating the testing process can significantly reduce testing cost, minimise human error and make regression testing easier [39].

The idea of the automatic generation of test cases has been applied to a broad variety of testing approaches. Such testing approaches can be classified in many different ways. The most general categories include static and dynamic testing, white-box and black-box testing, and functional and non-functional testing. The basic notions underlying these varied perspectives, along with some other specific testing techniques, i.e. temporal testing and stress testing, are briefly summarised as follows:

2.2.1 Static and Dynamic Testing

In some parts of the literature, software testing is categorised by whether the programmed code is executed or not [39]. *Static testing* is a type of testing that is performed without executing the program. It includes software inspections and some forms of analysis. By reviewing the documents (e.g. requirements, design and test cases), the software inspections can be used to find and eliminate errors or ambiguities in such documents. The types of reviews in formality order range from informal, walkthrough, peer review and inspection, respectively. There are also some forms of analysis, such as static program analysis, model checking and formal method.

Dynamic testing executes the program with real inputs (i.e. a given set of test cases) and checks the expected outputs. Nowadays, the term 'testing' used in most of the literature refers to dynamic testing, whereas static testing is often called 'verification activities' [39].

2.2.2 White-Box and Black-Box Testing

Another traditional and fundamental classification of software testing is known as the *box approach*. The box approach is based on the point of view that a test engineer takes when designing the test process (e.g. test cases). It is broadly divided into two primary methods: white-box and black-box testing. Though some literature additionally includes grey-box testing [45], which is a combination of those two methods, into the classification, only white-box and black-box techniques are briefly described here.

For *white-box testing*, an internal or structural view of the SUT is mainly concerned. Implicit knowledge of the system's inner workings is therefore required to conduct the testing [4]. Sometimes, it is also referred to as *structural testing* [45, 4]. As stated above, exhaustive testing is impractical, in order to verify the structure of the program, the test suite needs to satisfy one or more coverage criteria [39]. So that, a particular set of elements, such as statements, branches, paths and internal logic of the code, should be executed by the test suite once [43]. Some main coverage criteria include statement coverage, branch coverage and condition coverage [4]. Although white-box testing can uncover many errors or problems in the SUT, it is often unable to detect missing or incorrect functionalities [4].

Black-box testing, by contrast, mainly concerns with a functional or external view of the item tested [4]. It is also referred to as *functional testing* or *behavioural testing* [45, 4]. Black-box testing validates whether a given system conforms to its specifications; it thus solely focuses on the outputs generated in response to the selected inputs and execution conditions (i.e. ignoring the internal mechanism of the system) [4]. Any specific knowledge of the underlying system is not necessarily required. Furthermore, system functionality faults can be uncovered by black-box testing.

2.2.3 Functional and Non-Functional Testing

Another software testing classification is based on what the software property is tested, i.e. functional requirements and non-functional requirements. As mentioned in Section 2.2.2, *functional testing* is sometimes also called 'black-box testing'. It tends to verify a specific action or function of the code. In particular, the input is fed to the function, and the output is examined.

Non-functional testing tends to verify the non-functional requirements of the software, such as scalability (or other performance issues), behaviour under certain constraints and security. These non-functional requirements reflect the quality of the product.

2.2.4 Temporal Testing

In addition to those previous testing categories, there are several specific testing techniques for their particular objective. *Temporal testing*, for instance, is used to verify the temporal behaviour of real-time systems [4].

In particular, correct functioning of the system, especially for safety-critical systems, such as avionics and automotive systems, relies on logical correctness as well as temporal correctness [5, 4]. Timing issues are essentially required to be verified, since violating timing constraints—either outputs produced too early or too late—may be fatal to human life [3].

Timing analysis is generally used as a means to verify the temporal behaviour of real-time systems owing to the fact that it is the foremost timing-related safety evidence [6]. Typically, in order to ensure that a system meets its timing requirements, the WCET of each task is determined, and then the Worst-Case Response Time (WCRT) of the task set is computed by using the WCETs as its inputs [6]. Note that WCRT analysis is out of the research scope; for further details, the reader may refer to [46, 47].

Unfortunately, determining *exact* WCET is infeasible since a certain variation of execution times of each task depends not only on the input data, but also the different behaviour of the environment [7], which may be influenced by its technical characteristics, such as the use of parallelism, distribution and fault-tolerant mechanisms [4].

Furthermore, such dynamic behaviour can exacerbate the WCET determination by the use of multicore platforms [7, 6], as briefly described in Section 2.1.4 above. As a result, either an *upper bound* or an *estimate* of the WCET is used for safety assurance instead [7, 6].

In practice, there are several approaches to determine WCET (for further details on an overview of various timing analysis techniques, as well as a variety of available commercial WCET tools, the reader may refer to [7]). In brief, however, determining the WCET can be classified into three major types, including static (analysis-based), dynamic (measurement-based) and hybrid approaches [6].

Static Methods Static WCET analysis techniques typically compute *upper bounds* on the execution time of a task by analysing the set of possible control flow paths through the task with no need of executing code on real hardware or on a simulator [7]. To obtain the upper bounds of WCET, the static analysis may combine the control flow with some annotations (i.e. abstract model of the hardware architecture) [7]. In particular, control-flow analysis and bound calculation are used to cover all possible execution paths, and abstraction is used to cover all possible context dependencies in the processor behaviour [7].

By producing bounds, static approaches can guarantee that the execution time will not exceed these bounds, and such bounds also allow safe schedulability analysis of hard real-time systems [7]. Furthermore, the approaches do not require any complex equipment to simulate the hardware and peripherals of the target system since they can be done without running the program [7]. However, processor-specific models of processor behaviour may necessarily be required for processor-behaviour analysis, and in many cases the obtained results are imprecise, e.g. overestimated WCET bounds [7].

Dynamic Methods Dynamic WCET analysis techniques do not produce bounds like static methods; rather they produce *estimates* by executing all possible execution paths of a task (or task parts) on the given hardware or simulator for some set of inputs [7]. In addition, instead of using processor-behaviour analysis, measurements are used for dynamic approaches [7]; the approaches are thus also called 'measurement-based methods' [7, 6]. During execution of the task, the measured times are taken and then the maximal observed execution times or their distribution are derived (or the measured times of code snippets are combined in order to result for the whole task) [7].

Dynamic approaches are simpler to apply to new target processors since processor behaviour is not required to be modelled [7]. Also, WCET estimates produced by dynamic timing analysis are more precise than the bounds from static approaches (i.e. closer to the exact WCET), especially for complex processors and complex application [7].

However, dynamic methods may be unsafe if all possible execution paths are not measured as some context-dependent execution-time changes may be missed [7]. Hence, the approaches may be useful for non-hard real-time systems (i.e. applications that do not require guarantees) because they may give the developer a feeling about the execution time in common cases and the likelihood of the occurrence of the worst case [7].

Dynamic timing analysis is also known as *temporal testing*, where its ultimate goal is to find test inputs that will cause the system to violate performance timing requirements [4, 9]. There are diverse ways of performing temporal testing of embedded real-time systems, for example, constrained random-based temporal testing, stress-based temporal testing, search-based temporal testing and mutation-based temporal testing [4]. In order to identify temporal failures in embedded or complex systems, it is generally agreed and several researchers also show that the last two techniques are substantially more effective than the first two techniques, which yet are easier and relatively inexpensive to implement [4]. Search-based temporal testing, which will be presented in the next section (Section 2.3), is the main focus of this research.

Hybrid Methods Hybrid approaches are timing analysis techniques that calculate an estimate of the WCET by combining static program analysis techniques and execution time measurements [7, 6].

Figure 2.6 illustrates a summary of relevant WCETs and Best-Case Execution Times (BCETs) of a real-time task.

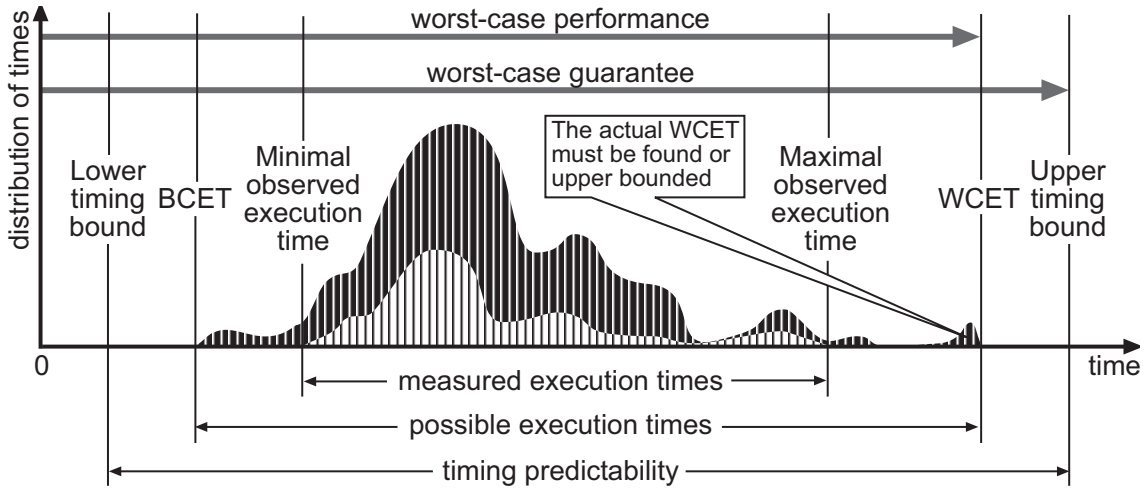


Figure 2.6: Basic notions concerning system timing analysis (reprinted from [7])

In particular, as mentioned earlier in this section, an execution time of the task can be varied by the input data and different behaviour of the environment, the upper curve (the darker one) represents the set of all execution times of the task [7]. The lower curve (the lighter one) depicts a subset of measured executions. So that, in practice, a WCET estimate (or the maximal observed execution time) obtained from dynamic approaches tends to underestimate the actual WCET [9]. The upper timing bound calculated by static analysis is often conservatively overestimate the actual WCET [9].

In addition, regarding the above testing categories (Sections 2.2.2 and 2.2.3), temporal testing can be classified as a type of (black-box) environment testing [4] since temporal behaviour, which is affected by the system environment, is assessed. Besides, temporal testing can also be considered as a non-functional testing [10] because time behaviour is one of the non-functional properties, which is defined in the international standard for the evaluation of software quality (ISO/IEC 25010:2011) [48].

2.2.5 Stress Testing

Another example of a specific testing technique is *stress testing*. For stress testing, the system is stressed beyond its normal operating conditions in order to validate its robustness and elasticity requirements [4]. In particular, the system should degrade gracefully rather than fail catastrophically when it is overloaded (called 'elasticity'), and it should also fully recover when the unrealistic load is removed (called 'robustness') [4]. Temporal testing is similar to stress testing in the way that they are used to test non-functional properties by attempting to stress the SUT. However, they differ in that temporal testing rather verifies the temporal quality of embedded systems, while stress testing validates the system's robustness and elasticity [4].

2.3 Search-Based Software Testing

As mentioned in the previous section (Section 2.2), software testing is an essential part for all software development to produce high-quality software [39]; however its tasks (e.g. test case generation) are extremely laborious, difficult and costly in practice [43, 40]. Automation of test activities, therefore, has often been viewed as a means of reducing its cost and improving its effectiveness [39, 40].

Nevertheless, software systems have become more and more complicated. In particular, software components may be developed by different vendors, using different techniques in different programming languages and even running on different platforms [40]. It is, therefore, more difficult to generate appropriate data in any reasonable time-frame for larger and more complex software [40].

Since previous efforts have been limited by the size and complexity of the software involved, together with the basic fact that test data generation is an undecidable problem, metaheuristic search techniques have been applied to the area of software testing, and this has given rise to the research topic known as SBST [49]. Optimisation algorithms are used in SBST to automate the search for test data that maximises the achievement of test goals while minimising testing costs [40]. Search-based approaches have been developed to address a wide and diverse range of domains, including testing approaches based on agents, aspects, interactions, integration, mutation, regression, stress and web applications [40]. The primary concern in all SBST approaches is to define a fitness function (or a set of fitness functions) that captures the test objectives and can be used to guide a search-based optimisation algorithm, which searches the space of test inputs to find those that meet the test objectives [49].

The next two sections (i.e. Sections 2.4 and 2.5) provide an overview of metaheuristics and hyper-heuristics, which are heuristic search methods for solving computational search problems.

2.4 Metaheuristics

Optimisation metaheuristic algorithms are broadly recognised as the efficient approaches for many hard optimisation problems [50]. The success of applying a metaheuristic on a given optimisation problem chiefly relies on how well it can provide a balance between the exploration (diversification) and the exploitation (intensification) over its problem space [50]. In order to find the optimum of the search space, for example, making large (almost random) changes does best for some kinds of problems, while making small local greedy changes does best for others [51]. Particularly, *exploration* is the ability of an algorithm to identify parts of the search space with high-quality solutions [50], e.g. being able to jump to the other regions [52]. *Exploitation*, on the other hand, is the ability of the algorithm to intensify the search in some promising areas of the accumulated search experience [50], e.g. being able to perform a local search within a limited region by using accumulated experience [52].

Metaheuristics can be classified in numerous ways based on their features or aspects. Examples of metaheuristic classifications include trajectory methods and discontinuous methods, population-based and single-point search, memory usage and memoryless methods, one and various neighbourhood structures, dynamic and static objective function, and nature-inspired and non-nature inspiration [53].

With regard to the classification that distinguishes between single-solution based and population-based metaheuristics, some major metaheuristic algorithms, which are commonly used for optimisation problems (and also used in this work), are briefly described in the following sections.

2.4.1 Single-Solution Based Metaheuristics

Single-solution (or, in other terms, 'single-point' and 'single-state') metaheuristics, which are more exploitation oriented, improve and maintain a single solution [50]. Generally, these algorithms start with a single initial solution and iteratively move away from a candidate solution to its direct neighbourhood [50]. The neighbourhood of the candidate solution is the set of solutions that are slightly different from the candidate solution [54]. Some main algorithms belonging to this category are Random Search (RS), HC, SA and Tabu Search (TS) [55, 50, 51, 56].

Random Search RS is one of the simplest and easiest to implement optimisation algorithms [9]. It is the extreme one in exploration owing to that fact that it explores the search space by randomly selecting a candidate solution and evaluating the fitness of the solution (without guidance) until the optimum is found or termination criteria are reached [57, 51], as illustrated in Algorithm 1.

RS may be very poor at finding an optimal solution. For example, in test data generation (e.g. structural testing), the number of inputs covering a particular structural target may occupy a very small part of the overall search space [9], as depicted in Figure 2.7.

However, since RS does not take much effort to implement and its evaluations can be done fairly quickly, it is usually used as a reference to compare its performance with other more sophisticated techniques (for the same number of evaluations) in order to evaluate the efficiency of such techniques [57].

Algorithm 1: RS [51]

1: $Best \leftarrow$ some initial random candidate solution
2: **repeat**
3: $S \leftarrow$ a random candidate solution
4: **if** Quality(S) $>$ Quality($Best$) **then**
5: $Best \leftarrow S$
6: **end if**
7: **until** $Best$ is the ideal solution **or** we have run out of time
8: **return** $Best$

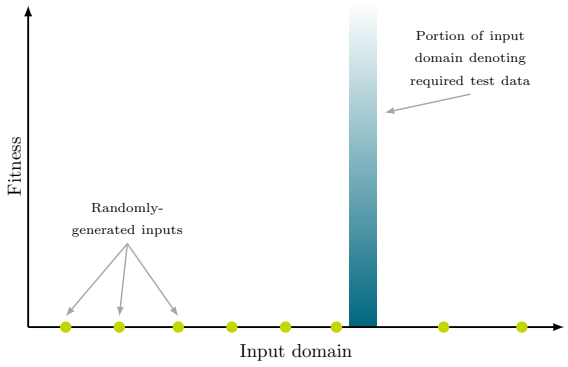


Figure 2.7: RS (adapted from [9])

Stochastic Hill Climbing HC, which is also referred to as a *local search*, is likely the oldest and simplest search algorithm that uses a *fitness function* to guide the search [55, 9]. A simple algorithm for HC is given in Algorithm 2. In particular, HC iteratively improves a candidate solution by replacing it with a better new solution, which is the one in its neighbourhood [51].

As depicted in Figure 2.8, it tries to climb up the hill. The algorithm, however, can easily be trapped in a local optimum, where the qualities of all neighbouring solutions are equal or worse [51].

Apart from the basic one mentioned above, in order to escape from the local optimum, some variant of HC, i.e. random-restart HC, may benefit from being able to restart and perform a climb from a new initial position in the landscape [9].

Some other, i.e. SHC, may be a little more aggressive as it explores a number of neighbouring solutions all at one time and then possibly adopts the best one [51], as presented in Algorithm 3.

Algorithm 2: HC [51]

```
1: S ← some initial candidate solution
2: repeat
3:    R ← Tweak(Copy(S))
4:    if Quality(R) > Quality(S) then
5:       S ← R
6:    end if
7: until S is the ideal solution or we have run out of time
8: return S
```

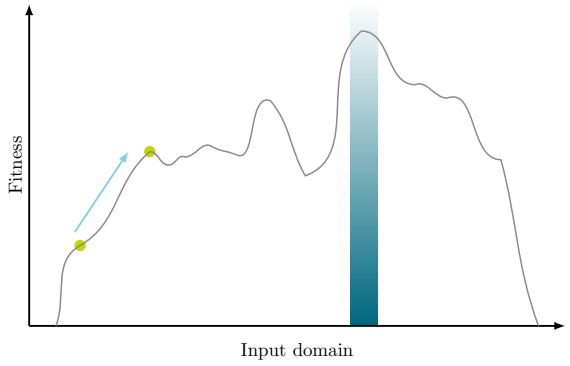


Figure 2.8: HC (adapted from [9])

Algorithm 3: SHC [51]

```
1: n ← number of tweaks desired to sample the gradient
2: S ← some initial candidate solution
3: repeat
4:    R ← Tweak(Copy(S))
5:    for n − 1 times do
6:       W ← Tweak(Copy(S))
7:       if Quality(W) > Quality(R) then
8:          R ← W
9:       end if
10:   end for
11:   if Quality(R) > Quality(S) then
12:      S ← R
13:   end if
14: until S is the ideal solution or we have run out of time
15: return S
```

Simulated Annealing SA is an alternative to the simple HC [9, 51]. It differs from HC (Algorithm 2) in its decision of when to replace the original candidate solution (S) with the newly tweaked child (R) [51]. In particular, if R is better than S, S will always be replaced with R as usual; but if R is worse than S, S may still be replaced with R with a certain probability $P(t, R, S)$ [51]:

$$P(t, R, S) = e^{\frac{\text{Quality}(R) - \text{Quality}(S)}{t}} \tag{2.1}$$

where $t \geq 0$. In other words, SA attempts to escape local optima by going down hills sometimes, as illustrated in Figure 2.9.

Regarding the probability $P(t, R, S)$, as presented in Algorithm 4, the solution R will be accepted, depending on t and on the values of the objective functions for R and S [50]. A control parameter (i.e. temperature) t regulates the probability P to accept the solutions by starting its initial value with a high number and then gradually decreasing itself towards the end of the search process according to the *cooling schedule* [58, 59]. The algorithm was inspired by the physical process of annealing in materials, i.e. the cooling of a material in a heat bath [58].

Algorithm 4: SA [51]

$t \leftarrow$ temperature, initially a high number
$S \leftarrow$ some initial candidate solution
$Best \leftarrow S$
repeat
  $R \leftarrow$ Tweak(Copy(S))
  if Quality(R) $>$ Quality(S) **or**
  a random number chosen from 0 **to** 1 $< e^{\frac{\text{Quality}(R)-\text{Quality}(S)}{t}}$ **then**
    $S \leftarrow R$
  end if
  Decrease t
  if Quality(S) $>$ Quality($Best$) **then**
    $Best \leftarrow S$
  end if
until $Best$ is the ideal solution **or** we have run out of time **or** $t \leqslant 0$
return $Best$

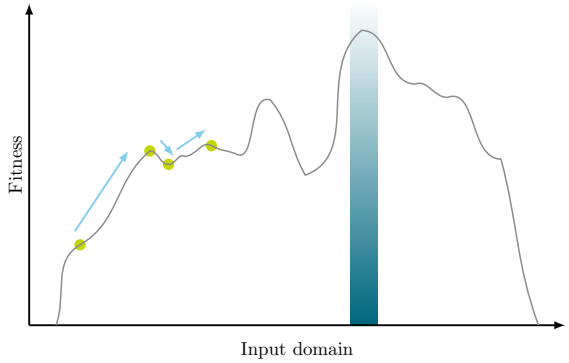


Figure 2.9: SA (adapted from [9])

Tabu Search TS uses the history of the search as a means to escape from local minima and to implement an explorative strategy [50]. Specifically, the idea is to prevent the cyclic repetition of recently visited solutions by maintaining a memory, which is called 'tabu list', and consequently to force the search to accept even deteriorating moves [50]. A simple TS algorithm with short-term memory is presented in Algorithm 5.

There are various types of memory structures used to remember specific properties of the recent search trajectory; these include short-term, intermediate-term and long-term memory structures [50].

For example, in *short-term memory*, the length of the tabu list controls the way of exploration, i.e. the search will concentrate on small areas of the search space if the length is low, and it will be forced to explore on larger regions if the length is high [50].

In some cases, the tabu list can prevent attractive moves or may lead to an overall stagnation of the search process, an *intermediate-term memory* (called 'aspiration criteria') can be applied and greatly improves the search process by overriding tabu restrictions with a set of rules [50]. Therefore, the aspiration criteria can allow a move, which is forbidden by the tabu list, to be revisited if it is satisfied [50].

In addition, the search can be allowed to avoid visiting solutions that present the most often encountered attributes or to visit solutions with attributes rarely encountered by using a *long-term memory*, such as a frequency memory [50].

Algorithm 5: TS [51]

$l \leftarrow$ desired maximum tabu list length
$n \leftarrow$ number of tweaks desired to sample the gradient
$S \leftarrow$ some initial candidate solution
$Best \leftarrow S$
$L \leftarrow \{\}$ a tabu list of maximum length l ▷ Implemented as first-in, first-out queue
Enqueue S into L
repeat
  if Length(L) $> l$ **then**
    Remove oldest element from L
  end if
  $R \leftarrow$ Tweak(Copy(S))
  for $n - 1$ times **do**
    $W \leftarrow$ Tweak(Copy(S))
    if $W \notin L$ **and** (Quality(W) $>$ Quality(R) **or** $R \in L$) **then**
      $R \leftarrow W$
    end if
  end for
  if $R \notin L$ **then**
    $S \leftarrow R$
    Enqueue R into L
  end if
  if Quality(S) $>$ Quality($Best$) **then**
    $Best \leftarrow S$
  end if
until $Best$ is the ideal solution **or** we have run out of time
return $Best$

2.4.2 Population-Based Metaheuristics

Population-based (or *multi-point*) metaheuristics commonly consist of a collection of individual solutions which are maintained in a population [50]. These optimisation algorithms are more explorative oriented as they sample many points in the search space at once. Yet, there are not simply just being a parallel hill climber because candidate solutions affect how other candidates will climb up the hill in the quality function and candidate solutions are forced to be tweaked in the direction of the better solutions [51]. Particularly, good solutions cause poor solutions to be rejected and then the new ones are created.

Evolutionary Computation Evolutionary Computation (EC), which is inspired by Darwin's evolutionary theory, is one of the most studied population-based methods [50]. A collection of EC algorithms is also known as Evolutionary Algorithms (EAs) [51, 50]. In the literature, a number of EAs have been proposed. Among them, the two most well-known ones, i.e. GA and Evolution Strategy (ES), are presented here. Other EAs include evolutionary programming, Genetic Programming (GP), estimation of distribution algorithms, differential evolution, coevolutionary algorithms, cultural algorithms, scatter search and path relinking [50].

Note: there are several other optimisation techniques, which are also classified as the population-based approaches. Those techniques are grouped under the term 'swarm intelligence', including ant colony optimisation, particle swarm optimisation, bacterial foraging optimisation algorithm, bee colony optimisation-based algorithms, artificial immune systems and biogeography-based optimisation [50].

Genetic Algorithm GA is a metaheuristic inspired by Darwinian principles of nature and the concept of survival of the fittest [9]. An *individual* (or a *chromosome*) refers to a candidate solution or a point in the search space currently under consideration. A set of individuals is called a 'population'. Generally, GA can also be classified as a generational algorithm, where the entire samples (individuals) are updated once per iteration [51]. Therefore, a *generation* denotes the population produced in each cycle by a selection mechanism and evolutionary operators (such as mutation, crossover and recombination) [51]. Figure 2.10 illustrates an example of GA, where individuals are generated in the search space at a time.

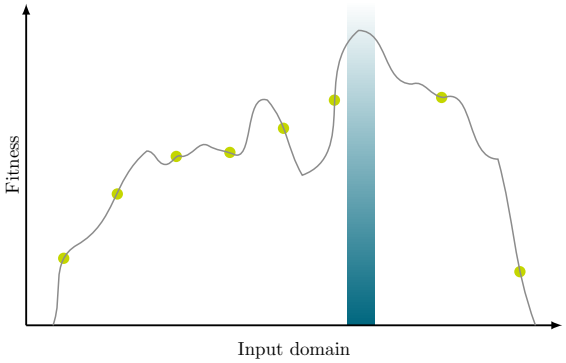


Figure 2.10: GA (adapted from [9])

In general, as presented in Algorithm 6, the GA process is started by randomly generating the first population in the initialisation stage [9]. Then, in the fitness evaluation phase, each individual is evaluated for fitness [9]. The *selection* mechanism decides which individuals should be parents for *crossover* [9]. After that, elements of each parent individual are combined to form two offspring individuals that embody characteristics of their parents [9].

Algorithm 6: GA [51]

```
popsize ← desired population size
P ← {}
for popsize times do
    P ← P ∪ {new random individual}
end for
Best ← □                                   ▷ The □ means nobody yet
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
        end if
    end for
    Q ← {}
    for popsize/2 times do
        Parent P_a ← SelectWithReplacement(P)
        Parent P_b ← SelectWithReplacement(P)
        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
        Q ← Q ∪ {Mutate(C_a), Mutate(C_b)}
    end for
    P ← Q
until Best is the ideal solution or we have run out of time
return Best
```

In order to diversify the search into new areas of problem space, the *mutation* mechanism is used to tweak some elements of the newly-created individuals at random, e.g. replacing one of such individual's elements with a newly-random generated element [9]. Finally, in the *reinsertion* step, the next generation of the population is chosen, and its individuals are inserted into a new round of selection for evaluating the fitness [9]. This cycle continues until the ideal solution is found or the allocated resources, such as a time limit or a certain number of fitness evaluations, are exhausted [9].

Evolution Strategy ES is another EA technique, which is very similar to GA. It differs from GA in that ES selects individuals by employing truncation selection and then breeds new individuals for the next generation by using the only mutation [51]. Examples of ES algorithms include (μ, λ)–ES and $(\mu + \lambda)$–ES.

Algorithm 7 shows pseudocode of one of the simplest ES algorithms called '(μ, λ)–ES algorithm'. Typically, the (μ, λ)–ES begins with a randomly generated population of λ number of individuals [51]. After that, the fitness values of all the individuals are assessed. Truncation selection is then utilised to keep the μ fittest individuals and discard the rest ones from the population [51]. As a parent, each of the μ fittest individuals produces λ/μ children through a mutation operator; so that λ new children are created [51]. For the next generation, the children replace the parents, who are deleted from the population [51]. In brief, the number of parents which survive for the next generation is μ, and the number of children that the μ parents make in total is λ [51].

The $(\mu+\lambda)$–ES algorithm, as listed in Algorithm 8, differs from the (μ, λ)–ES algorithm in that, rather than simply replacing the parents with the children in the next generation, $(\mu+\lambda)$–ES allows the parents to compete with the children to survive in the next generation [51]. Therefore, all successive generations consist of the μ parents plus the λ new children [51]. Since high-fitness parents persist to compete with the children, the $(\mu + \lambda)$–ES may be more exploitative than the (μ, λ)–ES [51].

Algorithm 7: (μ, λ)–ES [51]

$\mu \leftarrow$ number of parent selected
$\lambda \leftarrow$ number of children generated by the parents
$P \leftarrow \{\}$
for λ times **do**
  $P \leftarrow P \cup \{\text{new random individual}\}$
end for
$Best \leftarrow \square$
repeat
  for each individual $P_i \in P$ **do**
    AssessFitness(P_i)
    if $Best = \square$ **or** Fitness(P_i) > Fitness($Best$) **then**
      $Best \leftarrow P_i$
    end if
  end for
  $Q \leftarrow$ the μ individuals in P whose Fitness() are greater
  $P \leftarrow \{\}$
  for each individual $Q_j \in Q$ **do**
    for λ/μ times **do**
      $P \leftarrow P \cup \{\text{Mutate}(\text{Copy}(Q_j))\}$
    end for
  end for
until $Best$ is the ideal solution **or** we have run out of time
return $Best$

Algorithm 8: $(\mu + \lambda)$–ES [51]

$\mu \leftarrow$ number of parent selected
$\lambda \leftarrow$ number of children generated by the parents
$P \leftarrow \{\}$
for λ times **do**
  $P \leftarrow P \cup \{\text{new random individual}\}$
end for
$Best \leftarrow \square$
repeat
  for each individual $P_i \in P$ **do**
    AssessFitness(P_i)
    if $Best = \square$ **or** Fitness(P_i) > Fitness($Best$) **then**
      $Best \leftarrow P_i$
    end if
  end for
  $Q \leftarrow$ the μ individuals in P whose Fitness() are greater
  $P \leftarrow Q$
  for each individual $Q_j \in Q$ **do**
    for λ/μ times **do**
      $P \leftarrow P \cup \{\text{Mutate}(\text{Copy}(Q_j))\}$
    end for
  end for
until $Best$ is the ideal solution **or** we have run out of time
return $Best$

2.5 Hyper-Heuristics

Although those metaheuristics presented in Section 2.4 and other search techniques are successful in solving real-world computational search problems, they can often face some difficulties in terms of easily applying them to newly encountered problems or even new instances of similar problems [60]. These difficulties mainly arise out of the significant range of parameter or algorithm choices involved when using this type of computational search approaches, together with a lack of guidance as to how to select them [60]. The level of understanding of the scientific community is insufficient to facilitate simple choices of which heuristic approach to effectively use in which situation [60].

Moreover, state-of-the-art approaches tend to represent bespoke problem-specific (or made-to-measure) methods, which are expensive to develop and maintain [60]. These obstacles have motivated researchers to seek a better way for developing more generally applicable search methodologies [60].

Hyper-heuristics, therefore, have been proposed as a search method to automate the design of heuristic methods and to tune such methods for solving hard computational search problems [61]. Rather than trying to solve the problem directly, hyper-heuristics, which can be regarded as off-the-peg methods, attempt to find the right method or sequence of heuristics in a given situation [60]. In particular, hyper-heuristics operate on a search space of heuristics (or heuristic components), instead of operating directly on the search space of solutions to the underlying problem that is being addressed [60].

Above all, hyper-heuristic approaches aim to raise the abstraction level of tools to the point where the tools automatically configure themselves to produce the most effective search strategies [62]. Particularly, different problems may require different search approaches. It may not be immediately apparent that which approaches will work best at hand for the problem. This requires the production of higher-level *tactic* that explore and exploit a combination of lower-level heuristics to solve the problems [62].

2.5.1 Classification of Hyper-Heuristics

Burke et al. [60] proposed the general definition of the term 'hyper-heuristics' as 'a search method or learning mechanism for selecting or generating heuristics solve computational search problems'. As illustrated in Figure 2.11, Burke et al. [62] further categorised hyper-heuristic approaches into two primary types based on the methodology used to deal with low-level heuristics: heuristic selection and heuristic generation.

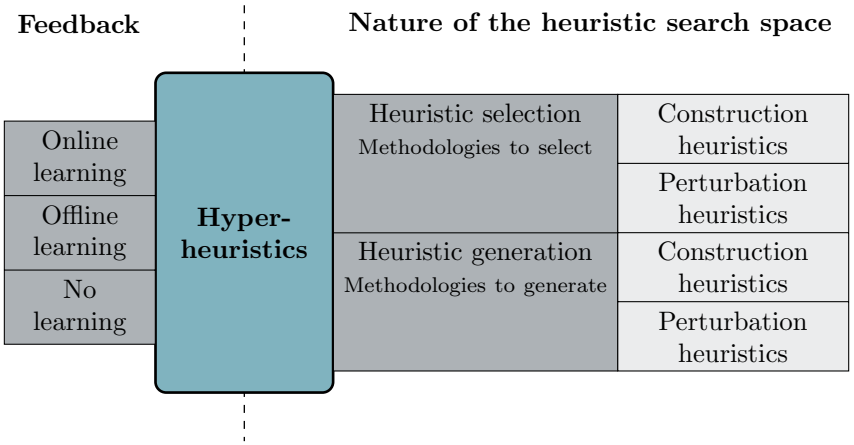


Figure 2.11: Classification of hyper-heuristic approaches (adapted from [62])

In particular, the *heuristic selection* is a methodology for choosing or selecting existing heuristics [62]. On the other hand, the *heuristic generation* is a methodology for generating new heuristics from the components of existing ones [62].

These two classes (i.e. heuristic selection and heuristic generation) can be further categorised into subcategories, depending on the nature of the low-level heuristics: constructive or perturbative search paradigms [62]. For *perturbative heuristic methods*, complete candidate solutions are considered, and these solutions are changed by modifying one or more of their components [62]. As opposed to the perturbation, *constructive heuristic methods* consider partial candidate solutions in which one or more components of such solutions are missing and iteratively extending them [62].

Additionally, as depicted in Figure 2.11, hyper-heuristics are able to be differentiated by sources of feedback information: online learning, offline learning and no learning [62]. Specifically, as a hyper-heuristic is considered to be a learning algorithm, some feedback information from the search (or learning) process is used for solving the underlying problems [62].

The *online* approaches take place while the algorithm is solving an instance of a problem [62]. The high-level strategy hence uses task-dependent local properties to determine the appropriate low-level heuristic to apply [62]. On the contrary, the *offline* methods gather knowledge from a set of training instances in the form of rules or programs that would hopefully generalise to the process of solving unseen instances [62]. *Non-learning* hyper-heuristics are those that do not use any feedback from the search process [62].

There have been numerous studies on hyper-heuristics as detailed in the recent survey by Burke et al. [60]. In this thesis, the heuristic selection is the main focus. In addition, since a candidate solution of the problem domain, i.e. temporal testing, is a test vector that has to be entirely executed with the SUT together at once in order to measure its execution time as a fitness function value, the perturbation is the appropriate preference category. Therefore, this research applies a *selection perturbative hyper-heuristic* to search for temporal test inputs that will produce extreme longest execution times, which may violate the system's temporal requirements.

2.5.2 Selection Perturbative Hyper-Heuristics

For selection perturbative hyper-heuristics, in general, low-level perturbative heuristics are selected to improve a candidate solution [63]. The selection perturbative hyper-heuristics can be separated into two major groups: *single-point* based and *multi-point* based search hyper-heuristics [63].

Single-point selection perturbative hyper-heuristics consist of two successive stages: *heuristic selection* and *move acceptance* [64]. An initial solution is generated, and then it is iteratively improved through these stages [52]. At each iteration, a new candidate solution is produced by selecting and applying a low-level perturbative heuristic from a predefined set of perturbative heuristics [52]. The methods, such as choice function, reinforcement learning, Monte Carlo methods and SA, have been used for heuristic selection [63]. Then, a move acceptance decides whether such the candidate solution should replace the incumbent solution [52].

Multi-point selection perturbative hyper-heuristics, on the other hand, consider populations of solutions. A number of metaheuristic techniques, including GA, great deluge, harmony search and TS, have been employed to explore the space of low-level perturbative heuristics [63].

2.6 Related Work

In software testing community, search-based approaches have been used across the spectrum of test case design methods. In particular, metaheuristic search techniques have been applied not only to white-box (structural), black-box (functional) and grey-box (combination of structural and functional) testing but also to non-functional properties.

Afzal et al. [10] systematically reviewed the research work published from 1996 to 2007 in the field of application of metaheuristic search techniques for testing five non-functional properties: execution time, Quality of Service (QoS), security, usability and safety. Accordingly, there are thirty-five primary studies related to the field: 15 (execution time), 2 (QoS), 7 (security), 7 (usability) and 4 (safety). A variety of metaheuristic search techniques, i.e. GA, SA, grammatical evolution (GE), GP and its variants, swarm intelligence methods, TS, HC and ant colony methods (AC), were found to be applicable for non-functional testing. Figure 2.12 illustrates the year-wise distribution of primary studies within each non-functional property as well as the frequency of application of different metaheuristics.

As shown in Figure 2.12, with respect to the studies on execution time, two well-known metaheuristic methods, namely GAs and SA, have been found efficacious in verification of timing-related constraints [10]. Those studies are briefly summarised in Table 2.5.

Figure 2.12: Distribution of SBST studies on non-functional properties over a range of metaheuristics and time period (adapted from [10])

Table 2.5: Summary of research papers on applying metaheuristics for testing temporal property

Year	Article	Metaheuristic	Aim	Fitness function	Benchmark(s)	Comparison	Findings
1996	Weneger et al. [65]	GA	WCET and BCET	Execution time (µs)	A simple C function	Statistical and systematic testing	1) the longest execution time of 26.27 µs was found quickly with GA in less than 20 generations; and 2) the shortest execution time of 5.07 µs, which is shorter than the shortest time determined so far by statistical and systematic testing, was found.
1997	Weneger et al. [66]	GA	WCET and BCET	Execution time (processor cycles)	Five test objects from different application domains	Random testing	GA consistently outperformed random testing by finding more extreme times.
1997	Alander et al. [67]	GA	WCET	Execution time (ms)	A power system protection relay software	Random testing	Based on the simulated environment, with the same amount of test cases, GA generated more input data cases with longer response times.
1998	Tracey et al. [68]	SA	WCET	Execution time (µs)	Four simple programs, varying in parameter spaces	The worst-case path of each program	For all test programs, each search resulted in a valid test case which exercised a worst-case path. The use of SA was more effective with larger, more complex, parameter spaces.
1998	O'Sullivan et al. [69]	GA with cluster analysis	WCET	Execution time (processor cycles)	A complex algorithm from automotive electronics domain	GA	The results showed that cluster analysis is a useful termination criterion as it provides detailed information about the convergence state of the evolutionary test. So that, a test is suggested to be terminated if it has converged to one or more local and global optima because the probability of finding even better solutions is small.

Continued on next page...

Continued from previous page...

Year	Article	Metaheuristic	Aim	Fitness function	Benchmark(s)	Comparison	Findings
1998	Weneger and Grocht-mann [70]	GA	WCET	Execution time (processor cycles)	Eight test objects from different application domains	Random testing	The experiments were extended from [66] by raising the range of input parameters to 5,000 and further included three more benchmarks. With a large number of input parameters, GA obtained more extremal execution times with less or equal test effort than random testing.
1998	Puschner and Nossal [71]	GA	WCET	Execution time (processor cycles only for `Heapsort` and abstract time units for the rest)	Seven programs	RS, best effort data generation and static WCET analysis	GA found same or longer times than random testing. In comparison with best effort timings, GA matched the timings and found the longer time in one case, while in comparison with static analysis, the upper bounds were not broken but were matched on several occasions.
1999	Pohlheim and Wegener [5]	Extended GA	WCET	Execution time (µs)	Bubble sort algorithm and software modules from a motor control project	Systematic testing	The results from bubble sort were used to find approximate evolutionary parameters for motor control software modules. Extended GA, which includes the use of multiple sub-populations (each using a different search strategy), found longer execution times for all the given modules in comparison with systematic testing.
2000	Groß et al. [72]	GA	WCET and BCET	Coverage of code annotations along shortest and longest execution paths	22 test objects, including simple sorting algorithms, modules from a graphical contour plotting package, modules from a robot vision system, line interpolation algorithms, modules from a train control systems and two artificial modules	All annotations which together make up the longest or shortest path for each test object	1) a prediction model based on the complexity of the test objects, which can be used to predict evolutionary testability; and 2) there are several properties inhibiting evolutionary testability: small path domains, high-data dependence, large input vectors and nesting.

Continued on next page...

Continued from previous page...

Year	Article	Metaheuristic	Aim	Fitness function	Benchmark(s)	Comparison	Findings
2000	Weneger et al. [73]	GA	WCET	Execution time (µs)	Six time-critical tasks in an engine control system	Developer-made tests	GA outperformed the developers' tests.
2001	Groß [74]	GA	WCET and BCET	Coverage of code annotations along shortest and longest execution paths	22 test objects	–	This work extended [72] by introducing source code measures with the intention to map program attributes which inhibit evolutionary testability into figures and establish a prediction system for evolutionary testability.
2001	Wegener and Mueller [75]	GA	WCET and BCET	Execution time (processor cycles)	Three real-time systems and two general-purpose algorithms	Static analysis	For WCET, the estimates of static analysis provided an upper bound while the measurements of evolutionary testing yielded a lower bound, and vice versa for BCET.
2003	Groß [76]	GA	WCET	Execution time (µs)	15 example test programs	Random testing and manual testing	Evolutionary testing outperformed random testing, i.e. random testing could only produce about 85 per cent of the maximum execution times found by evolutionary testing. The human tester was more successful in four out of 15 test programs. It indicated the presence of properties of test objects that inhibit evolutionary testability, i.e. the ability of an evolutionary algorithm to successfully generate test cases that violates the timing specification.
2005	Briand et al. [77]	GA	Missed deadline	Exponential fitness function based on the difference between executions deadline and executions actual completion	Two case studies, i.e. artefact scenarios and an actual real-time system	Task deadlines	GA could identify seeding times that stress the system to such an extent that small errors in the execution time estimates can lead to missed deadlines.

Continued on next page...

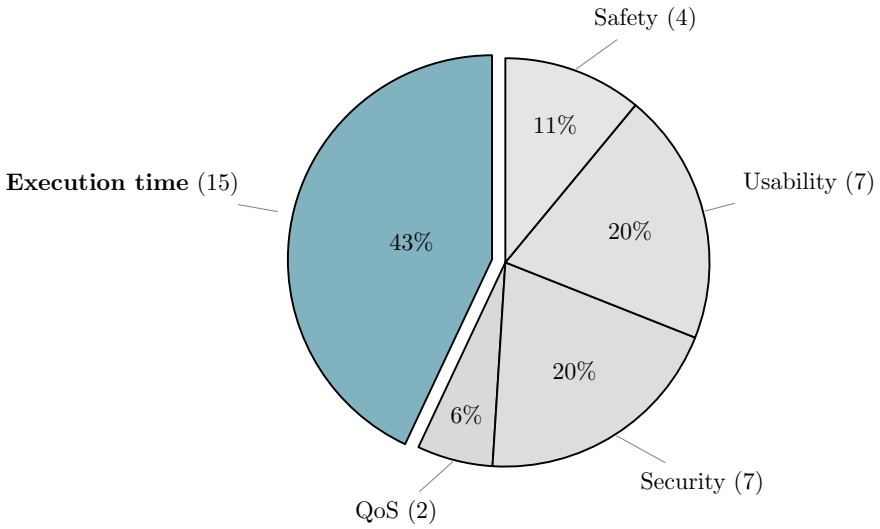
Continued from previous page...

Year	Article	Metaheuristic	Aim	Fitness function	Benchmark(s)	Comparison	Findings
2006	Tlili et al. [78]	Extended GA	WCET and BCET	Execution time (processor cycles)	12 test objects	GA	For almost all the test objects, an extended GA, which is seeded by test data that achieve a high structural coverage and restricts the range of input variables, outperformed (with fewer generations) standard evolutionary testing when measuring long execution times. Similar results were achieved for finding the shortest execution times.
2011	Bate and Khan [18]	GA	WCET	Execution time (processor cycles), loop count, instruction cache misses and data cache misses	16 programs from the Mälardalen WCET research group [79]	The highest-quality solution produced for the respective benchmark problems	1) a multi-criteria fitness function did not work well in practice; and 2) fitness heuristics were proposed as a guide to choosing criteria as a fitness measure for specific characteristics of the program.

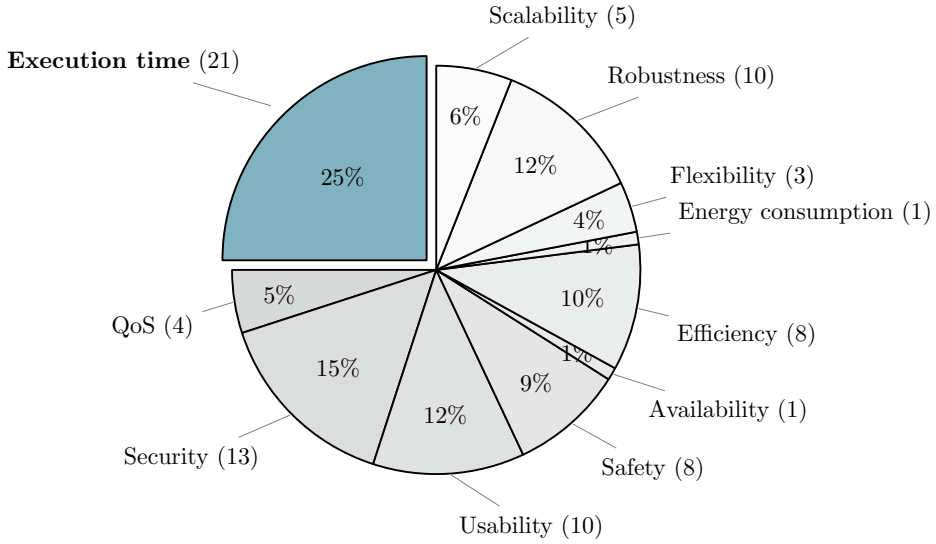
In short, almost all of these studies aimed to search for input situations that produce very long execution times (or WCETs) [65, 66, 67, 68, 69, 70, 71, 5, 72, 73, 74, 75, 76, 78]; some of them further included finding very short execution times (or BCETs) [65, 66, 72, 74, 75, 78]. In addition, a few of them also attempted to improve the performance of evolutionary testing by using particular strategies, such as the incorporation of cluster analysis [69], the use of multiple sub-populations (each using different strategy) [5], a prediction model for evolutionary testability [72, 74] and the use of high structural coverage test data as seeds and search space reduction [78].

Generally, the fitness function of temporal testing is the execution time, which is measured either by a system clock (such as in milliseconds (ms) or microseconds (µs)) or in terms of processor cycles, as depicted in Table 2.5. Afzal et al. [10] mentioned that measurement in terms of processor cycles is more popular because it is more precise and independent of the interrupts from the OS (e.g. context switching and paging). They also added that measurement in terms of processor cycles is deterministic in the sense that it is independent of system load and results in the same execution times for the same set of input parameters [10]. However, such measurement is dependent on the compiler and optimiser used, therefore the processor cycles differ for each platform [10]. We are rather interested in non-deterministic behaviour occurred in the multicore environment, therefore execution times are measured by a system clock in nanoseconds (ns) in this thesis.

Apart from the studies on temporal testing, where input data are normally generated and executed on a target task, GA was used for searching the sequence of arrival times of events for aperiodic tasks, which will cause the greatest delays in the execution time of the target task, as well [77]. Briand et al. [77] stated that both input data and seeding times for aperiodic tasks could impact on the execution time of the target task. Rather than simply finding temporal test inputs, they sought scenarios of event arrival times that would make the target task to miss its deadline and named the approach 'stress testing' [77]. Recall from Section 2.2.5 that the objective of stress testing is to validate the robustness and elasticity requirements of the system, but the objective of [77] was to find the sequence of arrival times of events for aperiodic tasks, which would cause the greatest delays in the execution of the target task.



(a) Categories of non-functional properties from 1996 to 2007



(b) Categories of non-functional properties from 1996 to 2014

Figure 2.13: Relative distribution of research papers on SBST for non-functional properties (adapted from [15])

A more recent survey of non-functional SBST research was conducted by Harman et al. [15]. The survey extended the work of Afzal et al. [10] by identifying six more non-functional attributes (i.e. availability, efficiency, energy consumption, flexibility, robustness and scalability) and including all the papers published from 1996 to 2014, as illustrated in Figure 2.13b. In particular, data from 1996 to 2007 (Figure 2.13a) were based on the systematic literature review by Afzal et al. [10], while data from 2008 to 2014 were collected from the SBSE repository [16].

With regard to the SBSE repository [16], where it is the first source of information on related work for the SBSE community [15], there have been a few recent studies on applying search-based approaches to verify temporal constraints that explicitly addressed the issue of interference introduced by multicore platforms. (The repository may not guarantee 100 per cent precision and recall, but it has proved sufficiently usable [15].)

For example, as briefly summarised in Table 2.5, Bate and Khan [18] investigated the effectiveness of GA in estimating WCET of software running on a modern processor, i.e. the ARM processor simulated on the SimpleScalar architecture [80]. Besides using the execution time as a fitness function alone, a number of program characteristics (e.g. loop iterations) and hardware features (e.g. branch prediction misses, data cache misses and instruction cache misses), were also examined their ability to supportingly guide the search to find the WCET. These dominant factors are believed to make finding the WCET on modern processors more difficult [18]. The approach is called a 'multi-criteria heuristic function' [17]. The results showed that although these factors, as well as a combination of them with the execution time, did not perform well in practice, they can still be gainfully used if appropriate ones are chosen [18]. As a result, fitness heuristics were proposed to suggest that which criteria should be used as the fitness measure for the particular program's characteristics [18]. For instance, if the program has a single path through it, then execution time should be used as the single fitness function [18].

Note that in order to obtain the execution time, which might be affected by *actual* interference that can take place within a multicore architecture, all the experiments in this thesis were run on the real hardware, i.e. the QorIQ P4080 multicore processor.

2.7 Summary

In this chapter, some features of multicore processor systems, such as shared caches, were highlighted and discussed their ability to cause interference which makes temporal behaviour verification more difficult. Nevertheless, it is unavoidable to ignore such verification, as violations of timing constraints of safety-critical systems may, in particular, be fatal to human life. A number of approaches to verify timing-related constraints were then addressed, especially search-based temporal testing, where optimisation technique is used to generate test inputs which will cause the system to violate performance timing requirements. This dynamic WCET analysis method will mainly be concerned in this research because it is simpler than the static WCET analysis method in terms of applying it to

new target processors, i.e. no specific details of a processor is required for its behaviour analysis. After that, various search-based optimisation algorithms, which are different in their ways of balancing between the exploration and exploitation, were briefly described. Finally, a survey of the relevant literature on using search-based approaches to verify timing constraints was provided.

Since there is a lack of guidance on how to select a metaheuristic technique to a particular situation, as previously mentioned in Section 2.5; rather than evaluating the effectiveness of an individual metaheuristic approach in the same way as those in Section 2.6, e.g. a comparison between GA and static testing in [75], this study will compare the effectiveness of applying diverse metaheuristics on temporal testing of a task running on a multicore platform. In the next chapter, therefore, both single-solution based metaheuristics (i.e. RS, HC, SHC and SA) and population-based metaheuristic (i.e. GA), which were given in Section 2.4, will be explored in details.

Chapter 3

Direct Optimisation

3.1 Introduction

3.1.1 Motivation

In the preceding chapter, a number of metaheuristic optimisation techniques were presented (Section 2.4), and their effectiveness in verifying temporal constraints was evidenced in the literature (Section 2.6). However, there is no rule of thumb on how to choose a particular approach for a particular (or instance) problem [60]. A metaheuristic approach can be successful in providing a 'sufficiently good' solution to an optimisation problem only when it is able to excellently balance exploration and exploitation over such problem space [50]. Furthermore, based on the previous work as summarised in Table 2.5, a certain metaheuristic algorithm (primarily the GA) was solely applied and its effectiveness was compared with other timing analysis techniques, such as static analysis, random testing and systematic testing; none of the work considered assessing the effectiveness among the metaheuristics themselves. In addition, only a few studies [17, 18] of search-based temporal testing explicitly emphasised multicore environments.

In this chapter, therefore, we explore how effective each metaheuristic algorithm is in finding test cases to exhibit the execution time of a task running on an embedded multicore platform.

3.1.2 Contributions

The contributions in this chapter are:

- Empirical evidence to demonstrate that metaheuristic search approaches are effective

ways of reaching extreme execution times of numerical functions running on an embedded multicore system;

- Empirical evidence to show that shared resources within a multicore environment genuinely impact on the temporal behaviour of a real-time embedded system when the problem size is too small.

3.1.3 Chapter Outline

The remainder of this chapter begins with a description of an experimental framework (Section 3.2), which was used throughout the thesis to facilitate the empirical study.

In order to assess the effectiveness of metaheuristic approaches, this chapter, in particular, separates the problem domain of temporal testing into two problem instances: 1) verifying the temporal behaviour of a single-threaded application; and 2) verifying the temporal behaviour of a multi-threaded application. Accordingly, the empirical work in this chapter considers five SUTs, including one single-threaded routine and four multi-threaded routines, as detailed in Section 3.3.

A preliminary analysis is provided in Section 3.4 to explain how specific parameter values are chosen for conducting the experiments throughout the chapter.

Experiment I (Section 3.5) demonstrates the performance of each metaheuristic technique for extremal timing performance of a single-threaded numerical function.

Experiment II (Section 3.6) demonstrates the performance of each metaheuristic technique for extremal timing performance of a multi-threaded numeric function.

In Section 3.7, the outcomes of the research described in this chapter are discussed and used to motivate the research in the subsequent chapter of this thesis.

3.2 Experimental Framework

A variety of metaheuristics, including RS, HC, SHC, SA and GA, are empirically investigated to determine their ability to find test inputs that will produce extreme (maximal) execution times when executed on a multicore chip. In order to complete the experiments, three elements are required to configure an experimental framework: a metaheuristics toolkit; a hardware platform on which to execute tasks and time those executions; and an interface between the two prior elements.

3.2.1 Metaheuristics Toolkit

The empirical studies on direct optimisation (in this chapter) and indirect optimisation (in the next chapter) were facilitated by the Java-based Evolutionary Computation Research System (ECJ) [81]. ECJ is one of the most popular EC toolkits, is extensible, and has a clear descriptive manual and strong community support (via a mailing list [82]) [83]. ECJ provides efficient implementations of a variety of EC approaches, such as GA, ES, GP, differential evolution and other population-based metaheuristics (e.g. particle swarm optimisation) [81]. Single-state based approaches, such as HC, SHC and SA, have recently been included in ECJ's latest version[1] [84].

However, all experiments in this chapter (and in the succeeding chapter) were performed using the ECJ version 23, which is the latest release available at the time of configuring the experimental framework. Accordingly, the population-based features of this ECJ version were adapted for the experiments of the single-state approaches. In particular, a simple HC and SA could be considered as the degenerate cases, i.e. $(1 + 1)$–ES, of the more general $(\mu + \lambda)$–ES [51]. Further details on modifying the ES features for HC, SHC and SA to generate temporal test inputs are as follows.

Candidate Solution A candidate solution for the problem of temporal testing is defined as a set of test inputs (or parameter values) in the form of a vector (sometimes simply called a 'test vector') for a function being tested. The search space of the problem is, therefore, the set of all possible test vectors for such the function.

Stochastic Hill Climbing Recall that, in Section 2.4.1, a new candidate solution, which is one in the neighbourhood of a current solution, is evaluated and it will be adopted if it is better than the current one for an HC algorithm. In this case, it could be thought of as a $(1 + 1)$–ES, where only one parent (i.e. the current solution) survived and one child (i.e. the new candidate solution) that the parent makes in total.

In the case of temporal testing here, a *neighbourhood* of an individual was defined as a candidate solution that is *randomly slightly different* from the current one. Particularly, only one position (or dimension) of a test vector changes to potentially any value (not just to 'nearby' values, as is more usual in HC) from a range of possible input values.

[1] The latest release (as of September 2017) is ECJ version 25, which was released on 9 July 2017 [82].

Therefore, the `ec.vector.VectorMutationPipeline`, which is the ECJ's common mutator for vector individuals, was used to generate the neighbourhood. In addition, in order to generate a child, which is slightly different from its parent, the mutation probability (P_m) was set to 0.1. The P_m is a probability of mutation per gene of an individual; it is not a probability of the whole individual [85]. An example of the neighbourhood generated by `VectorMutationPipeline` is shown in Figure 3.1.

Steepest Ascent Hill Climbing Recall also that SHC is a little more aggressive version of HC. A number of candidate solutions are created all at one time in each iteration, and then the best one is adopted if it is better than the current solution. In this case, it could also be thought of as a $(1 + \lambda)$–ES, where only one parent survived and λ children that the parent makes in total.

A *neighbourhood* of an individual for temporal testing, in this case, was defined as a set of candidate solutions, where only one position within a vector of each candidate (i.e. a child) is *slightly different*, either by adding or subtracting a value within the position with a fixed delta value (δ) from the current one (i.e. the parent).

In other words, the neighbourhood is a change of one allele by plus or minus delta, and all these possible neighbours are examined. So, the number of neighbourhoods (or λ) are double the length of the current solution. The neighbourhood here is very much smaller than in the simple HC case.

Figure 3.2 depicts an example of 14 children of a parent with a length of seven genomes, and the δ is given as five. In this regard, `SHCVectorMutationPipeline` was additionally implemented as a breeding pipeline for generating children, corresponding to the aforementioned neighbourhood's definition.

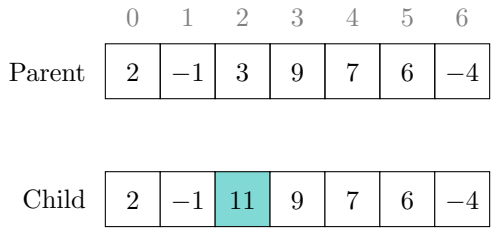


Figure 3.1: Example of the neighbourhood generated by `VectorMutationPipeline` mutator for the test vector length of 7

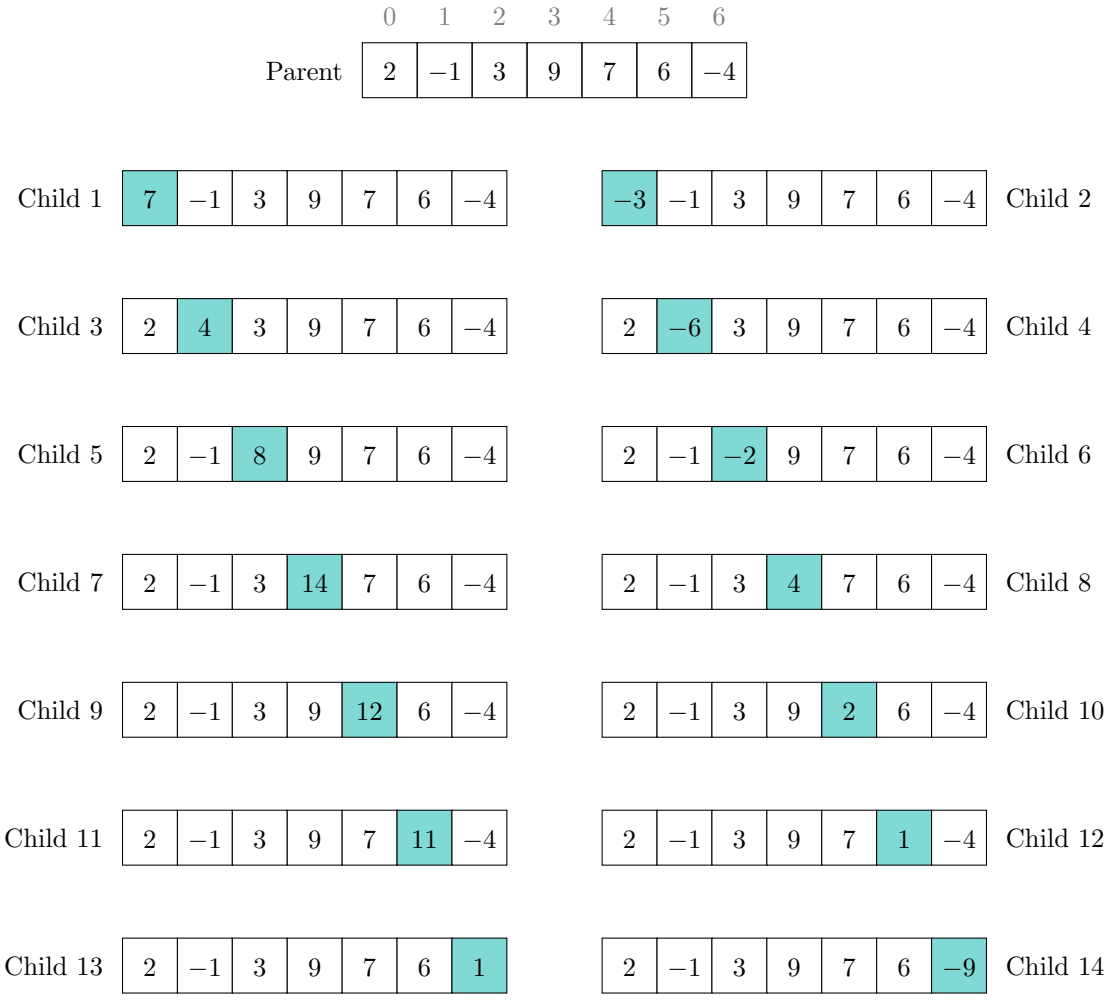


Figure 3.2: Example of the neighbourhoods generated by `SHCVectorMutationPipeline` breeding pipeline for the test vector length of 7 with $\delta = 5$

Simulated Annealing SA is quite similar to the basic HC, but some worse neighbours may be accepted and replaced the current candidate with a certain probability. Likewise, SA could be thought of as a $(1 + 1)$–ES.

Accordingly, a special selection operator, `SAESSelection`, was additionally implemented to accept some worse children during the search process. Also, the geometric reduction cooling function [58, 59], $T_{k+1} = \alpha T_k$, was used because it is the most common scheme used in the SA literature and is normally used as a baseline for comparison with other more elaborate schemes [86].

The parameters for SA were as follows: the sequence index in temperature cycle $k \in \mathbb{N}$, the initial temperature $T_0 = 1{,}000$ and cooling rate $\alpha = 0.99$. In fact, the value of α is typically in the range of $0.9 \leq \alpha \leq 0.99$ [86]. Since the number of iterations for SA in this

research is high (which will be described later in Section 3.5.2), we, therefore, chose 0.99 as the α value for the moderately slow cooling rate.

3.2.2 Timing Hardware Platform

For the purpose of verifying temporal behaviour on a real-time embedded system, and a safety-critical system in particular, in a multicore environment, the benchmarks, which will be presented later in Section 3.3, were run and their execution times were measured on a FreeScale QorIQ P4080 processor (P4080). The experiments were conducted on the P4080 processor owing to a wide variety of applications of the P4080 in industrial real-time embedded systems [87], such as telecommunications and networking [28], as well as on safety-critical systems, such as aerospace and defence markets [88].

The P4080 processor includes eight e500 PowerPC cores scaling to 1.5 GHz, and has a three-level cache-hierarchy: 32KB I/D L1, 128KB private L2 per core and 2 MB shared L3, as previously illustrated in Figure 2.2. In particular, the COMX-P4080 COM Express Module (COMX-P4080) [89, 90], which is a Single Board Computer (SBC) that the P4080 processor is embedded in a plug-in COM Express® module[2], was used as a multicore platform in this study. The block diagram of the COMX-P4080 development board is illustrated in Figure 3.3. (Note: there are several SBCs that provide a high-performance alternative to the COMX-P4080 on the market, such as the P4080 development system (P4080DS) [91] and the P4080 processor-based conduction- or air-cooled 6U cPCI Module (XCalibur1600) [92].)

In terms of software, the COMX-P4080 development system contains U-boot, Mentor Embedded Linux®, GNU Compiler Collection (GCC) tools (i.e. compiler and debugger) and CodeWarrior evaluation copy [89]. Unfortunately, there is no Java Runtime Environment (JRE) available for the development board, but the ECJ is a Java-based toolkit as previously described. In fact, although the full GCC suite is available, which means Java source code can be compiled using GNU Compiler for Java (GCJ), the GCJ is no longer maintained and will not be part of future releases [93]. Therefore, in order to conduct the experiments, an interface between the toolkit and the multicore platform was configured instead.

[2]COM Express is a Peripheral Component Interconnect (PCI) Industrial Computer Manufacturers Group (PICMG) standard for a Computer-on-Module (COM) form factor with PCI Express interconnects [89].

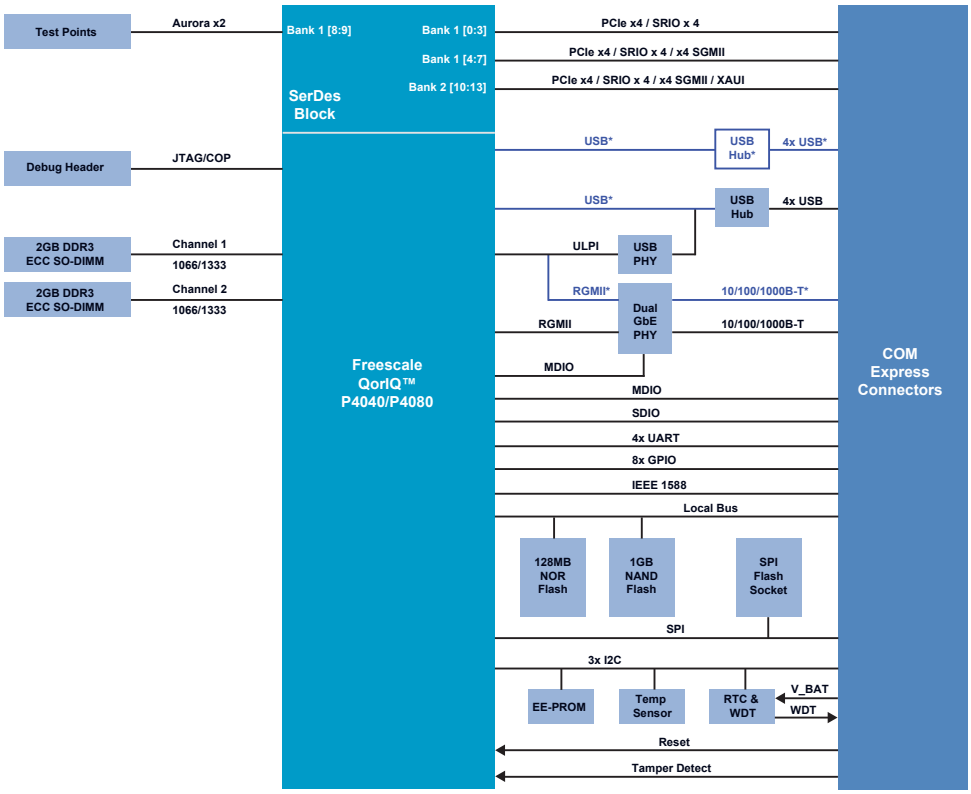


Figure 3.3: Block diagram of COMX-P4080 COM Express module (reprinted from [94])

3.2.3 Interface

The experimental framework was established by using the client-server model architecture to facilitate the communication between the workbench, where the ECJ toolkit is running on, and the COMX-P4080 development board, as depicted in Figure 3.4.

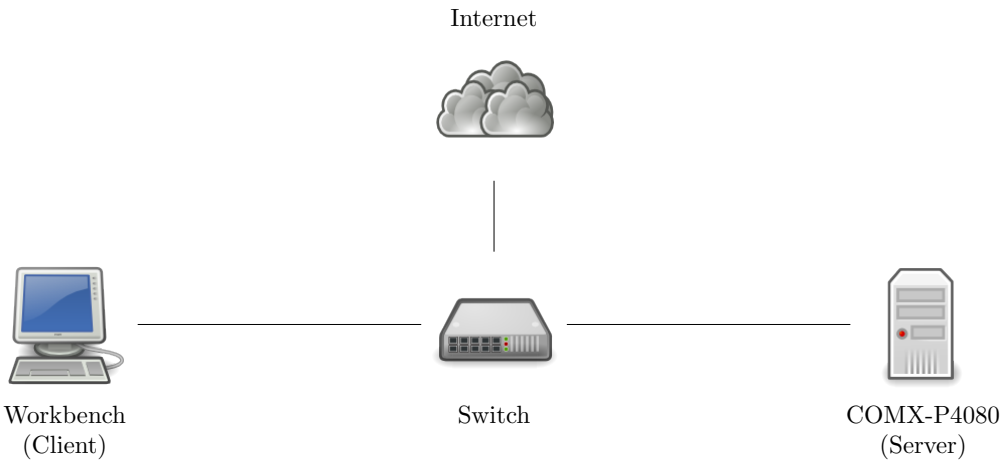


Figure 3.4: Overview of experimental framework

In particular, the workbench was set as a client, and the COMX-P4080 was set as a server. The workbench is the Intel® Core™ i7-5960X processor[3] with a RAM size of 64 GB. These two machines were connected together with a network switch in order to enable them to receive, process and forward data between each other.

On this computer network, some further configurations are required. Specifically, the File Transfer Protocol (FTP), which is a standard network protocol [95], was used for transferring the files between the client and server. Furthermore, the Secure Shell (SSH), which is a cryptographic network protocol [96], was used as a secure channel for remotely operating the server by sending shell commands from the client site. Additionally, in order to implement the FTP and SSH on the client site, the singleton pattern, which is a software design pattern that restricts the instantiation of a class to one object [97], was also applied to create a connection to the server only once during the entire experimental period. By using this design pattern, a connection error due to an auto disconnection by the COMX-P4080, which occurs when there are continual attempts by the client to reconnect the server, can be avoided.

An overview of the experimental procedure is illustrated in Figure 3.5. In particular, the iterative process starts with generating a test case, which is a sequence of integer arguments, by the metaheuristic toolkit running on the client site. The test case is then written into a *.csv file, e.g. `input.csv`. After that, this input file is transferred to the server through the FTP. Next, in order to obtain a fitness function, which is an execution time of a benchmark, the client sends a shell command to operate the server by way of the SSH to execute such test case with the benchmark.

After the server executed the test case for a number of times, a median from these runs is written into an output file, e.g. `output.csv`. (Further details on the number of runs and median will be given in Section 3.4.1.) Finally, the output file is requested by the client via the FTP for evaluating the fitness function. During the search process, the fitness function is used to guide the searches and eventually to find an optimum; in this optimisation problem, the maximum one is desirable. This process repeats until the termination condition of a search algorithm satisfied. The number of generations is given as the termination condition for all search algorithms researched in this thesis.

[3]The i7-5960X processor features a 3GHz clock speed, 20MB of L3 cache and eight physical processor cores.

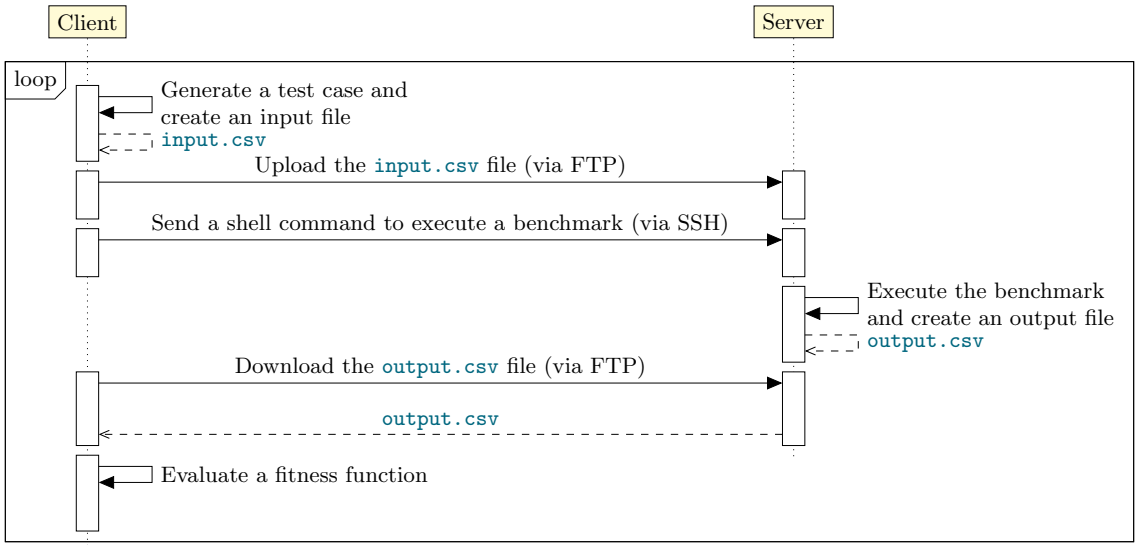


Figure 3.5: Sequence diagram of experimental procedure

Moreover, since the tasks naturally take a very tiny amount of time for computation, their execution times are therefore captured in ns by using function `clock_gettime` with a clock source `CLOCK_MONOTONIC` in order to get more precise and accurate timing. Also, this clock source is not affected by changes in the system time-of-day clock, while the time from a clock source `CLOCK_REALTIME` may leap forward or even backward after a time adjustment.

3.3 Software Under Tests

This section describes numerical functions, which were used as the benchmarks for the experiments of this chapter and the rest of the thesis. The numeric functions are the main concern in this research for the reason that they are essentially used in most applications to perform basic numerical calculations or even used as elements of more complex mathematical computations. A number of commercial and academic institutions have facilitated the applications by providing such numerical functions on scientific libraries. Some of the libraries are the GNU Scientific Library (GSL) [98] for C language, and Apache Commons Mathematics Library (Commons Math) [99] and Flanagan's Java Scientific Library [100] for Java language. The libraries generally offer a wide range of mathematical routines, such as complex numbers, roots of polynomials, sorting, statistics and some special functions, such as exponential and trigonometric functions.

Based on the multicore environment, in this chapter, both single-threaded (or sequential) and multi-threaded (or parallel) computational routines, i.e. root finding of polynomials and sort routines, respectively, were used for assessing the effectiveness of metaheuristics for generating test inputs giving rise to extreme execution times.

3.3.1 Polynomial Root-Finding Algorithm

GSL[4] [98] is the free software provided by GNU operating system (GNU). Linux OS, which is one of the GNU versions, is nowadays widely used on servers, personal computers, supercomputers, mobile phones and network routers [101], as well as the P4080 platform. All GSL functions are thread-safe, as a result, they can be used in multi-threaded programs [98]; albeit they are implemented in a single-threaded procedure.

There are several functions for evaluating and solving polynomials given by GSL [98]. A function for finding roots of a general univariate polynomial equation in the form of $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$, where the coefficient of the highest order term must be non-zero, `gsl_poly_complex_solve`, was chosen for this study because it allows us to explore the temporal behaviour with varying integer input sizes. By using balanced-QR reduction of the companion matrix to compute the roots of the general polynomial, this root finding routine returns `GSL_SUCCESS` if all the roots are found; otherwise, the error handler is invoked with an error code of `GSL_EFAILED` [98] (as shown in the code fragments of Appendix A.1).

In particular, the routine finds all of the real roots of a given polynomial by using an iterative method, i.e. the QR algorithm, to compute the eigenvalues of a companion matrix [102]. The reason is that, in the context of linear algebra, the eigenvalues of a matrix are considered (approximately) equivalent to the roots of the characteristic polynomial [103]. There are two main steps involved to determine such eigenvalues: 1) forming the companion matrix associated with the input polynomial, and then 2) computing the eigenvalues of this non-symmetric matrix by using a matrix balancing technique and followed by the QR iteration method [104].

[4]The GSL used in this thesis is version 2.1, while the current one (as of September 2017) is version 2.4 (released on 19 June 2017).

QR Algorithm As the name suggests, the basic idea of QR method is to perform a QR factorisation (also called a 'QR decomposition'), which is a technique to decompose a (complex) square matrix A into a product of an orthogonal matrix Q and an upper triangular matrix R [105]:

$$A = QR \tag{3.1}$$

In linear algebra, a matrix Q is orthogonal if its transpose is equal to its inverse, i.e. $Q^T = Q^{-1}$ [105]. It is also necessary that matrix Q is invertible ($Q^{-1} = Q^T$), unitary ($Q^{-1} = Q^*$) and therefore normal ($Q^*Q = QQ^*$) in the reals [105]. Accordingly, matrix A can be transformed as follows [106]:

$$A = RQ = Q^{-1}QRQ = Q^{-1}AQ = Q^TAQ \tag{3.2}$$

By reversing the order of multiplication product of Q and R and eliminating R, thus:

$$RQ = Q^*AQ \tag{3.3}$$

and since Q^*AQ is a similarity transformation of A, RQ has the same eigenvalues as A [107]. Based on this principle, the QR method computes the eigenvalues of matrix A by generating a sequence of matrices A_k initiated with $A_0 = A$ and given by:

$$A_k = R_kQ_k \tag{3.4}$$

where Q_k and R_k represent the QR factorisation of A_{k-1} [107], so that:

$$A_{k-1} = Q_kR_k \tag{3.5}$$

This process repeats for k-th iterations until the matrix R_kQ_k converges to an upper triangular matrix, such that we can eventually read off the eigenvalues from the diagonal [107].

Worst-Case Analysis According to the QR decomposition, different coefficients might require a different number of iterations for the routine to find the roots of a corresponding polynomial equation. As a result, the root-finding routine takes different times on different coefficient vectors. The worst-case scenario of polynomial root-finder can happen when a particular coefficient vector causes the greatest number of steps required for convergence of the QR algorithm. Note that the QR function presented in Listing A.2 of Appendix A.1 does not always guarantee convergence as the maximum number of iterations is given at 120, if any further iteration is required, the error handler will be invoked to inform that the QR reduction does not converge.

3.3.2 Sorting Algorithms

Over approximately 80 per cent of all processing cycles are accounted for by sorting [108]. A sorting routine is required in many applications, such as presenting the results from database queries, compiling a list of business investments with associated risk-reward measures and figuring the company payroll [108]. Although much processing time nowadays is spent on graphical interfaces, visualisation processing and video games, sorting remains a vital part of the computation [108].

Sorting functions are also usually included in libraries. For example, the GNU C library (glibc), which is the core library function for GNU/Linux systems, provides `qsort` (quicksort algorithm) as a standard library function for sorting [109], whereas GSL uses the heapsort algorithm for all its sorting functions (e.g. `gsl_heapsort` and `gsl_sort`) [98]. Quicksort and heapsort are commonly used by the libraries because they are asymptotically efficient with average time complexity (and also worst-case complexity) $O(n\,log\,n)$.

In this research, however, a number of Pthreads-based sorting algorithms used in the courses relevant to parallel programming from academic institutions were chosen. (Algorithmic efficiency is not a requirement.) The parallel sorting routines include bubble sort [110], shell sort [111], quicksort [111] and merge sort [112].

These parallel sorting algorithms, which sort numbers in ascending order, are designed based on general-purpose decomposition techniques, i.e. data decomposition and recursive decomposition, to split the computations to be performed into a set of tasks for concurrent execution. It is, however, not always guaranteed that a given decomposition can lead to the best parallel algorithm for a given problem [113].

Bubble Sort An example of a bubble sort used as a benchmark here is implemented in a parallel manner by using a data decomposition design on the odd-even transposition sort algorithm as presented in Listing A.3 of Appendix A.2.

Generally, an odd-even transposition sort, which is an exchange sort related to bubble sort, functions by comparing adjacent pairs of items and exchanging them if they are found to be out of order relative to each other [114]. However, during different phases of the sort, rather than one at a time comparing adjacent elements in the same way as in simple bubble sort, the odd-even transposition sort compares disjointed pairs by using alternating odd and even index values [108], as illustrated in Figure 3.6.

In particular, decoupling the compare-swaps is the key to the algorithm, i.e. all of the compare-swaps in a single phase (either the odd or even phase) can concurrently occur [114].

In a parallel version of the odd-even sort, data decomposition can additionally be applied to the sorting algorithm by dividing the array into chunks in order to form independent data chunks for threads [108]. The number of chunks directly depends on the given number of threads as every single thread is assigned to a particular chunk.

Shell Sort A concurrent shell sort illustrated in Listing A.4 parallelises the sorting by modifying the sequential algorithm code to be able to sort an h-partition all at once.

A sequential shell sort typically breaks the original array into a number of smaller sub-lists, each of which is then sorted by using an insertion sort [108]. In particular, instead of breaking the array into sub-lists of contiguous items, each sub-list is created by using an increment h (sometimes also called a 'gap' or 'interval' or 'partition') to choose all items that are h items apart [108].

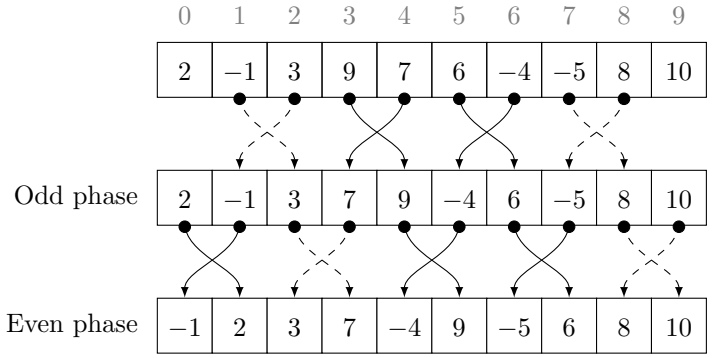


Figure 3.6: Serial odd-even transposition sort

Figure 3.7 shows an example of a shell sort with an increment of three ($h = 3$). Accordingly, there are three sub-lists (i.e. $N/h = 10/3 = 3$) for this pass, and then each of the sub-lists can be sorted by the insertion sort.

After completing insertion sorts in the first pass, shell sort algorithm continues the process by decreasing the value of h on every consecutive pass until $h = 1$, where it becomes a standard insertion sort.

In practice, an initial gap should start out much larger for larger arrays [115]. However, it is difficult to decide which the 'perfect' gap sequence to use since if there are too few gaps, it will slow down the passes, whereas if there are too many gaps, it will produce an overhead [115].

In Listing A.4, the sorting uses a gap sequence based on the reverse form of Knuth sequence, which is generated by the recursive expression $h = 3 * h + 1$, where the initial value of h is 1 [116]. Also, it takes advantage of data decomposition by independently doing an insertion sort on each sub-list of an entire h-partition before going to the next pass [108].

Quicksort A parallelising divide-and-conquer algorithm of quicksort, which is presented in Listing A.5, deploys a recursive decomposition design to divide the problem into a set of independent sub-problems, each of which can be executed on different cores.

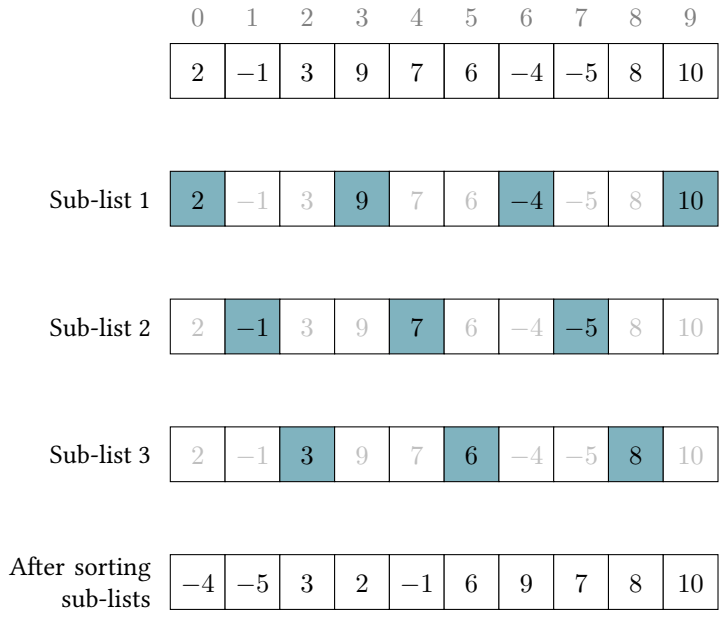


Figure 3.7: Serial shell sort with $h = 3$

A serial quicksort is naturally implemented as recursive procedures. It begins by choosing an element from the unsorted array as a *pivot* [108]. There are many different ways of picking the pivot and partitioning.

The algorithm's performance is greatly affected by the specific implementation scheme [117]. The algorithm in Listing A.5 is implemented based on Hoare's partition scheme [118]. In Figure 3.8, as well as in Listing A.5, for example, the first element is always picked as the pivot. Then, it partitions the given array around the picked pivot by moving elements to either side of the pivot, depending on the elements' relation to the pivot value [108]. The ones that are less than or equal to the pivot are on the left while the others are on the right. The above steps are recursively and separately applied to both sub-arrays until the base case of the recursion is reached, i.e. the arrays of size zero or one [116].

As shown in Listing A.5, the recursive decomposition is done by assigning an independent task to a specific thread. In particular, via a single-slot buffer, thread 0 works on a recursive partitioning task, while thread 1 does a sequential quicksort on those partitioned sub-arrays [119].

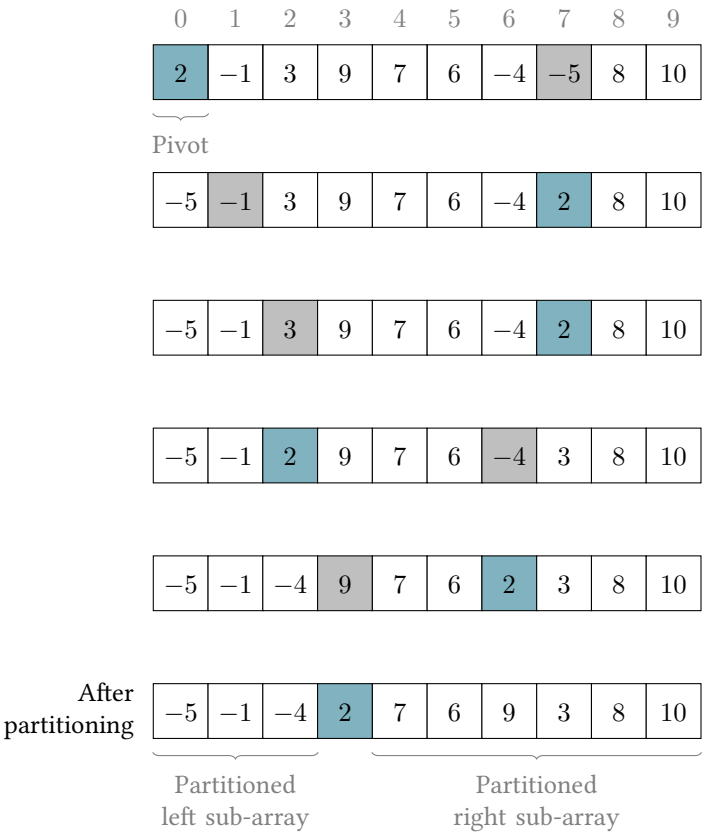


Figure 3.8: Serial quicksort

Merge Sort The parallelised merge sort (shown in Listing A.6) is based on a recursive decomposition design that recursively makes two child threads, i.e. one for the left half and another for the right half.

In the sequential version, based on a top-down implementation as depicted in Figure 3.9, merge sort recursively splits the unsorted list into sub-lists until sub-list size is one, which is considered sorted [117]. Then, it merges the sub-lists to produce newly sorted sub-lists until there is only one sub-list remaining [117].

For the concurrent version, every two threads are assigned to the available cores in order to perform merge sort for the left and right sub-lists as presented in Listing A.6.

Worst-Case Analysis Regarding the description of sorting algorithms above, the different numbers of comparisons and swaps may be involved to complete the sorting with a particular input list. In other words, it takes different execution times for variant lists. The worst-case scenario of these comparison-based sorting algorithms can occur when a particular order of elements requires the maximum number of comparisons and swaps to sort. The worst case of each parallel sorting is distinctive owing to its particular algorithm implemented.

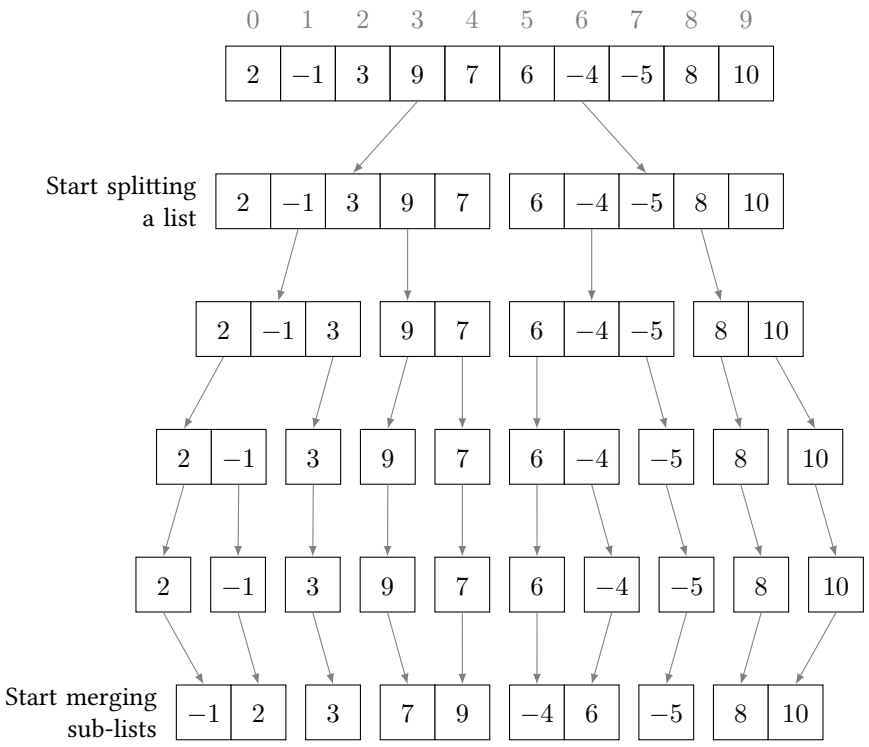


Figure 3.9: Serial merge sort

In bubble sort, for example, the input data that is in the reverse order will lead to the worst-case scenario [120]. In particular, starting with the highest value, the first element will be swapped with every other element down the list until it is in the last position and, consequently, $N - 1$ comparisons of adjacent values are required [120]. For the next successive passes, the number of comparisons is gradually reduced by one in each pass. The process repeats until the list sorted.

For shell sort, its performance not only relies on insertion sort used in each sub-list but also the choice of the increment sequence which affects the order of elements in each sub-list [120]. Therefore, the worst-case scenario of shell sort can arise when the input data is in the sequence in accordance with [121]. For instance, given $N = 8$ and $h = \{5, 3, 1\}$, so the worst-case permutation of the sorted list $\{1, 2, 3, 4, 5, 6, 7, 8\}$ would be $\{8, 5, 2, 6, 3, 7, 4, 1\}$, which takes the maximum number of 19 swaps [121], whereas the reverse order, i.e. $\{8, 7, 6, 5, 4, 3, 2, 1\}$, requires only 10 swaps.

In the case where a pivot is the first element of the list, the worst-case behaviour of quicksort would appear when such list is either already sorted or in the reverse order [120]. Since in each pass the pivot element is either the smallest or largest one, after partitioning the array, there is no element in one sub-array and $N - 1$ elements in the other [120]. As a result, at each recursive call, only one element could be removed from the list and consequently maximises the number of comparisons [120].

In case of merge sort, the worst-case scenario happens if, in every merging of sub-lists A and B, the elements of A and B are interleaved based on their value [120]. Since, in each comparison, one element from either A or B will be moved into a sorted list C, the total number of comparisons is $N_A + N_B - 1$. For example, the worst-case permutation of the sorted list $\{1, 2, 3, 4, 5, 6, 7, 8\}$ would be $\{5, 1, 7, 3, 6, 2, 8, 4\}$, which needs 17 comparisons, while the reverse order takes 12 comparisons.

For further details on sorting algorithm analysis, the reader may refer to [115, 120].

3.4 Preliminary Analysis

In the literature, it is broadly acknowledged that a good initial parameter setting potentially has more chance of being successful in applying a metaheuristic technique to a concrete problem [50]. On the other hand, parameter tuning is a tedious and time-consuming task [50]. To reduce such the task, several attempts have been made and are widely called 'automatic parameter tuning' [122] and 'adaptive metaheuristics' [50], which however are not the main objective of the research. In this study, therefore, the metaheuristic parameters were set to the values that are generally used and suggested by [51, 85]. Accordingly, Tables 3.2 and 3.3 list the parameter settings of metaheuristics used in this chapter, as well as in Chapter 4.

3.4.1 Number of Runs

Beside the metaheuristics' parameters, the *number of runs* is also an important one as it may affect the trajectory of the search. Particularly, in order to obtain a more precise fitness value of each test case, rather than using a benchmark's execution time from a single run as a cost function, each test case is instructed to repeatedly run with the benchmark for a number of times and *median* of these runs is instead used to represent the benchmark's fitness function. The median is used because it is not affected by extremely large or small on their values [123] (or, in other words, to eliminate noises from the collected data). For proof of concept, we simply need a reliable cost function. Thus, the median empirically determined measurement suffices. Other measures are not precluded.

We first performed a preliminary assessment to identify a proper number of runs to be used for the whole study. In this way, a GA was executed by using default parameter values (as presented in Table 3.2) and with four different numbers of runs (i.e. 50, 100, 1,000 and 10,000, respectively) to find a set of five coefficients that maximise the execution time of the polynomial root-finder for the general quartic equation.

The effect of taking a median from the different numbers of runs over 101 generations (i.e. a hundred generations plus an initial generation) of GA is illustrated in Figure 3.10. Note: each marker on a line represents the global best fitness value so far along the way of the search process.

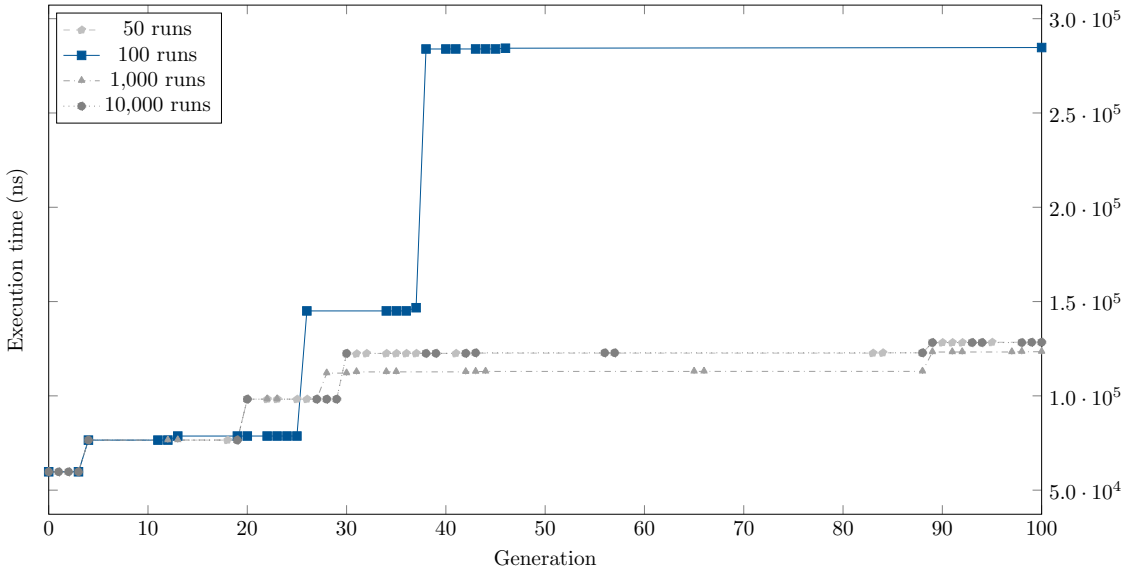


Figure 3.10: Results obtained by GA to the quartic equation with different numbers of runs

The results shown in Figure 3.10 indicate that a 100 is a proper number of runs to be used to get the median since it led the search to discover test vectors that produce the better result, i.e. a longer execution times of the polynomial solver, compared to others. Also, a hundred runs used a shorter computational time in comparison with 1,000 and 10,000 runs; it took a ten and a hundred times shorter, respectively.

3.4.2 Stopping Criterion

In addition to the number of runs, a *stopping criterion* is also an important factor that not only affects the search result but also directly affects the computational time used to perform the metaheuristic search algorithm. We conducted a preliminary analysis to determine a suitable value for the stopping criterion within an acceptable computational time, e.g. there is neither an improvement of the search process for a prolonged period nor a significant difference in the results compared to the time that the algorithm spent. In this research, the *number of evaluations* was used as a terminal condition.

To this end, we again initially run GA to generate test vectors of length five for a polynomial root finder with the same default parameter setting as in Section 3.4.1 (but with a different random seed). Since GA is a population-based approach, its number of evaluations is calculated by multiplying a population size by the number of generations.

However, rather than finding both the suitable population size and generations, we fixed the population size to 100 as it is sufficiently diverse for each generation of the GA search process to evolve its solutions in the population. We then only focused on determining an appropriate number of generations. In this manner, the number of generation for GA was set to 1,001 and subsequently the total number of evaluations is 100,100. The best fitness value of each generation is plotted in a graph as shown in Figure 3.11.

The graph in Figure 3.11 reveals that, during the search for finding temporal test vectors, GA was fluctuated at the beginning and then it was roughly stable after the generation of the sixties. Also, the global best fitness values (of approximately 83,500 ns) appeared between generations 61 to 63. After that, it jumped to the execution times at around 52,000 ns in the 95th generation and kept almost steady (as shown in a zoom-in view) until it slightly increased again to about 53,000 ns at the 861st generation.

Therefore, we decided to use 101 as the proper number of generations (or particularly 10,100 evaluations) because the global best fitness values were found in the early stage of the search, and there was very little progress after such generation.

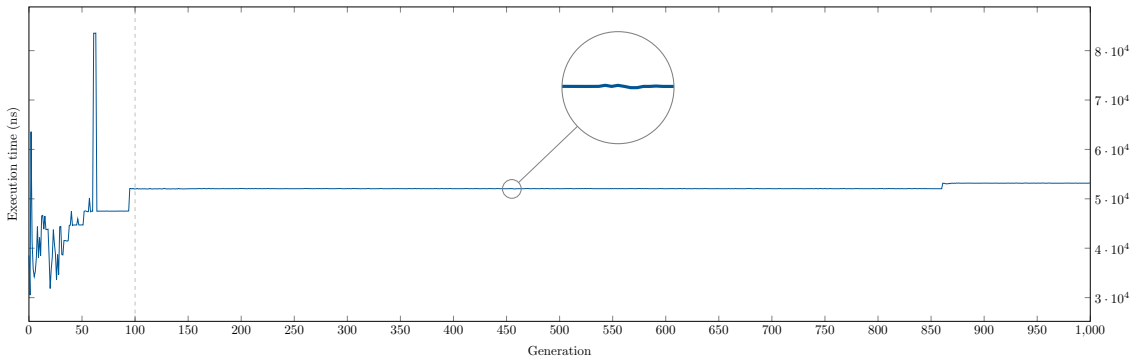


Figure 3.11: Best fitness values over 1,001 generations of GA

3.5 Experiment I—Single-Threaded Routine

3.5.1 Objectives

Although legacy code cannot take advantage of this computing power, it is interesting to examine the temporal behaviour of such sequential code on a multicore environment, as well as to assess the effectiveness of metaheuristics on this instance of temporal testing. In this experiment, we apply metaheuristic algorithms to search for test inputs that might cause a single-threaded routine, i.e. finding the roots of a polynomial equation, running on the COMX-P4080 board to violate performance timing requirements.

Note that the use of a single core of a multicore chip is actually considered a feasible proposition in some critical environments. When this approach is adopted, the aim is to make the execution as much like a single core chip as possible. Arguing the safety of such operation may well be simpler. (The application must, however, not need the enhanced compute power multicore use provides.) Therefore, the problem statement in this experiment is:

Problem Statement: *For a polynomial equation of the form* $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$*, we seek values of the coefficients (*a_0 *to* a_n*) that maximise the execution time of the polynomial solver.*

Accordingly, the research questions addressed are:

Research Question 1: *Among the metaheuristics, which technique is the best for seeking values of the coefficients that maximise the execution time of the polynomial solver?*

Research Question 2: *Are metaheuristics effective in seeking values of the coefficients that maximise the execution time of the polynomial solver?*

3.5.2 Preparation

In this chapter (and the rest of the thesis), the `int` type was primarily concerned for all experiments. The range of the input data was given, from −32,768 to 32,767; this equals to a 2-byte signed range of an integer data in C programming language. Note that the storage size for the `int` type on the COMX-P4080 development board is 4-byte though. The polynomial root finding algorithm was executed with 5, 7, 9 and 11 inputs (i.e. coefficients), respectively.

Since a representation used in this search problem is an integer vector, the number of all possible solutions of each problem input size is calculated by raising the total number of integers between −32,768 and 32,767 to the power of an input size: $65{,}536^n$, where n is the number of arguments or the size of integer inputs. The domain cardinality of each size of integer inputs is summarised in Table 3.1.

The algorithm parameter settings used in this experiment are listed in Tables 3.2 and 3.3. In particular, the population size was defined at 100 individuals over 100 generations (plus the initial generation) for GA. Some predefined genetic operators for GA, such as crossover and mutation, on the ECJ toolkit were also used.

In particular, the `VectorCrossoverPipeline`, which is a typical crossover operator for vector individuals, was used together with a uniform crossover[5] with the crossover probability (P_c) of 0.5. Every gene in a genome of the parents is crossed over independently with such certain probability. Besides, the `VectorMutationPipeline`, which is a common mutator for vector individuals, was used with the mutation probability (P_m) of 0.05. Each gene of an individual is then mutated with such probability.

Furthermore, in order to choose any two parents in the population for the crossover operator, a tournament selection was used as the primary selection technique for GA with a default tournament of size two [51, 85]. Elitism was also included to GA in order to allow the search to be more exploitative. The number of elite members was defined at 10 per cent of the population size (or ten individuals here). So, the best individuals from the previous population are kept around in future populations by directly injecting them into the next population [51].

Table 3.1: Domain cardinalities of the input arguments under consideration for the polynomial root-finding routine

Data type	Arguments	Input range	Cardinality
`int`	5	−32,768 to 32,767	$65{,}536^5$
`int`	7	−32,768 to 32,767	$65{,}536^7$
`int`	9	−32,768 to 32,767	$65{,}536^9$
`int`	11	−32,768 to 32,767	$65{,}536^{11}$

[5]The `VectorCrossoverPipeline` also supports other crossover types, such as one-point and two-point crossovers.

Table 3.2: Parameter settings for metaheuristic search algorithms

	Algorithm		
Parameter	**HC**	**SA**	**GA**
Generations	10,099	10,099	101
Population size	2	2	100
μ	1	1	-
λ	1	1	-
P_c	-	-	0.5
P_m	0.1	0.1	0.05
Tournament size	-	-	2
Elitism	-	-	0.1
Evaluations	10,100	10,100	10,100

Table 3.3: Parameter settings for SHC (Experiment I)

	Arguments			
Parameter	**5**	**7**	**9**	**11**
Generations	1,011	723	562	460
Population size	2	2	2	2
μ	1	1	1	1
λ	10	14	18	22
δ	50	50	50	50
P_m	0.1	0.1	0.1	0.1
Evaluations	10,102	10,110	10,100	10,100

In addition, as illustrated in Tables 3.2 and 3.3, the resource usage, i.e. timing spent for performing each algorithm, was controlled by specifying the parameters of algorithms in the way that they could take almost the same number of evaluations. Accordingly, the total number of evaluations for GA, which is 10,100 (as a result of the preliminary analysis in Section 3.4.2), was used as our desired amount of evaluations; this takes approximately three hours to complete its search process of each run.

In cases of HC and SA, therefore, the number of generations was given at 10,099. Based on their population size of two, i.e. one for a parent (μ) and the other one for a child (λ), their initial generation will perform two evaluations for their two initial random generated individuals. So that, one of them will later be a survived parent (or a current solution) for the next generation. Then, an evaluation will be performed once per generation to evaluate a newly generated child (or a candidate solution).

For SHC, however, the number of generations and λ are varied depending on the genome size (or a test input size) for a problem as shown in Table 3.3. Recall that, in Section 3.2.1, λ is double the size of the genome. Besides, in ECJ, a population size of $(\mu + \lambda)$–ES can be initialised for its initial generation with any size, and then the size will be automatically changed and equal to $\mu + \lambda$ for the next generations [85]. Also, an evaluation is required only once for each candidate solution. In addition, δ was given at 50, as we desired the SHC to be a bit more exploitative.

3.5.3 Method

In order to perform this empirical work, each metaheuristic algorithm was executed with each problem input size of the polynomial function ten times. We will refer to each execution of the algorithm as a *trial*. Each trial was provided with a different seed. In particular, the number of trials was specified to the ECJ's `jobs` parameter, and an initial seed, which is a Pseudorandom Number Generator (PRNG) obtained from `random.org`, was specified to the ECJ's `seed.0` parameter. Based on the job function of ECJ, the initial seed is given to the first job (`job.0`), and then the ECJ will automatically generate seeds for the rest (`job.1` to `job.9`) by incrementing the given initial seed. The initial seeds in this empirical experiment are summarised in Table 3.4.

Table 3.4: Initial seeds of metaheuristic algorithms for different input arguments of the polynomial root-finder (Experiment I)

	Arguments			
Algorithm	**5**	**7**	**9**	**11**
SHC	−16,424	−7,434	25,919	−20,920
HC	24,359	−18,192	11,984	20,253
SA	15,122	−20,872	−26,947	−6,722
GA	−25,510	22,960	31,954	−31,171
RS	12,301	15,625	28,866	23,782

3.5.4 Results

The results in this experiment are as follows:

From Figures 3.12 to 3.15, each bar chart shows the differences between the initial and the final (best) fitness values gained from ten trials of each metaheuristic algorithm on stressing the GSL's polynomial routine with the different number of input arguments, i.e. 5, 7, 9 and 11, respectively. Particularly, the darker bar represents an initial fitness, whereas the lighter bar represents the best fitness. The percentage improvement is given on the tip of the lighter bar.

In each box-and-whisker plot of Figure 3.16, a box depicts a distribution of the best fitness values obtained from ten trials of each metaheuristic approach to stress the polynomial solver with a particular number of input arguments. In particular, the bottom and top of the box are the first and third quartiles, and the band inside the box is the median. Also, the box's whiskers indicate variability outside the upper and lower quartiles, such as extreme fitness values.

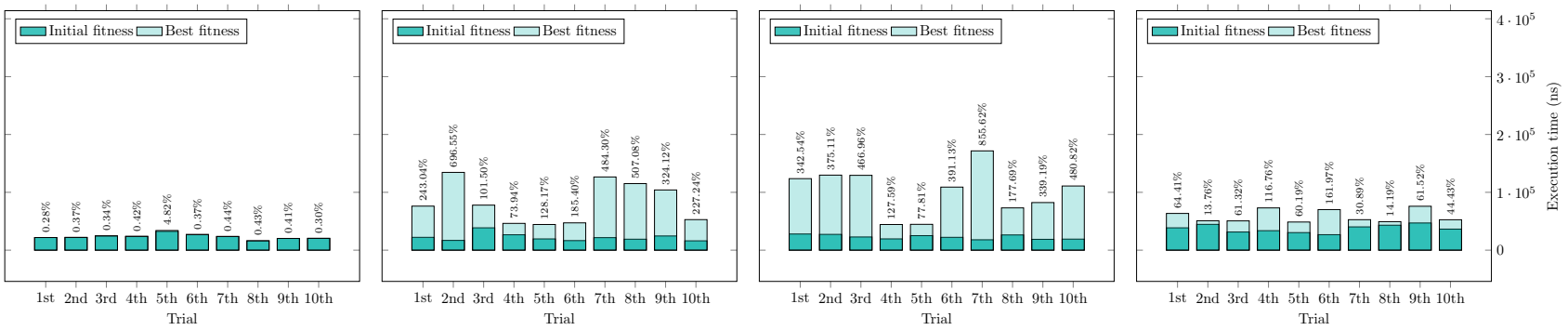
Table 3.5 summarises the most extreme execution time among ten trials of each algorithm. The result of the best performer for each problem input size is marked by a dagger (†), and the result of the worst is marked by an asterisk (∗).

3.5.5 Discussion and Conclusions

Research Question 1 In order to answer the first research question, we compared the effectiveness of the four metaheuristic methods on the ability to seek values of the coefficients that maximise the execution time of the polynomial solver.

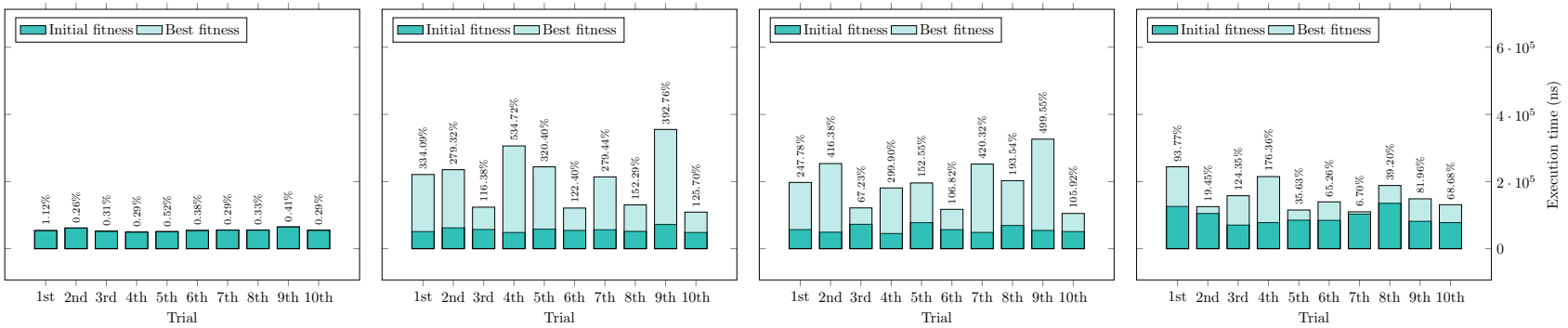
According to Figures 3.12 to 3.15, in comparison with other techniques, there was a very small improvement in finding temporal test inputs that maximise the execution time from SHC. Only a few trials clearly gave an improvement, such as the seventh trial of the problem size of 9, and the first, fifth and seventh trials of the problem size of 11. SHC seems to have difficulty escaping local optima.

To further describe the situation of getting stuck on local optima of SHC, let's consider Figure 3.12a, for example, where the first trial gave the lowest improvement rate at 0.28 per cent among the others, while the fifth trial made the highest improvement rate at 4.82 per cent. Accordingly, we plotted line graphs to display the global best fitness values from the beginning to the end of the SHC search process on these two trials as illustrated in Figure 3.17.



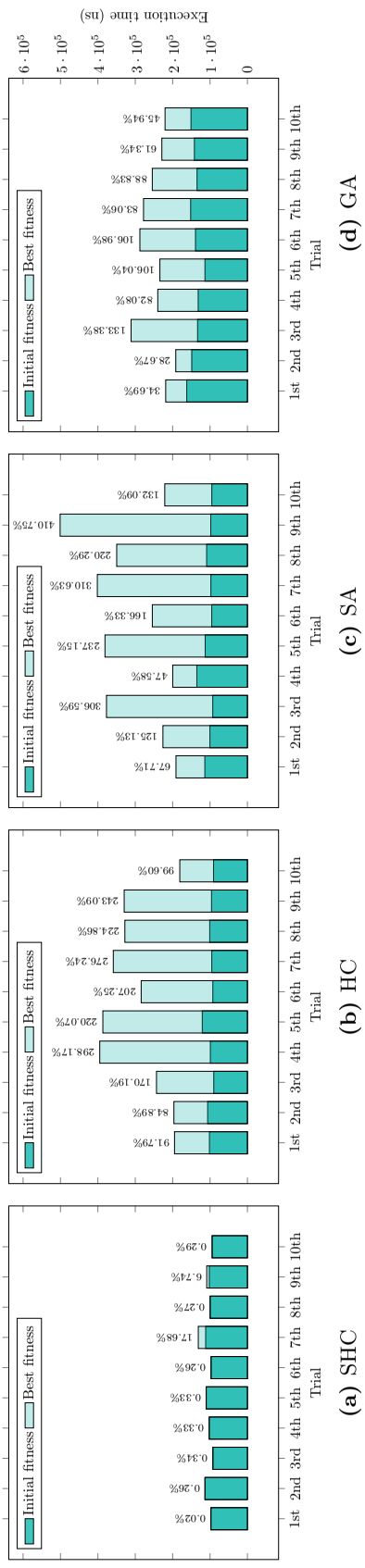
(a) SHC **(b)** HC **(c)** SA **(d)** GA

Figure 3.12: Results of 10 trials obtained by each metaheuristic algorithm to the quartic equation



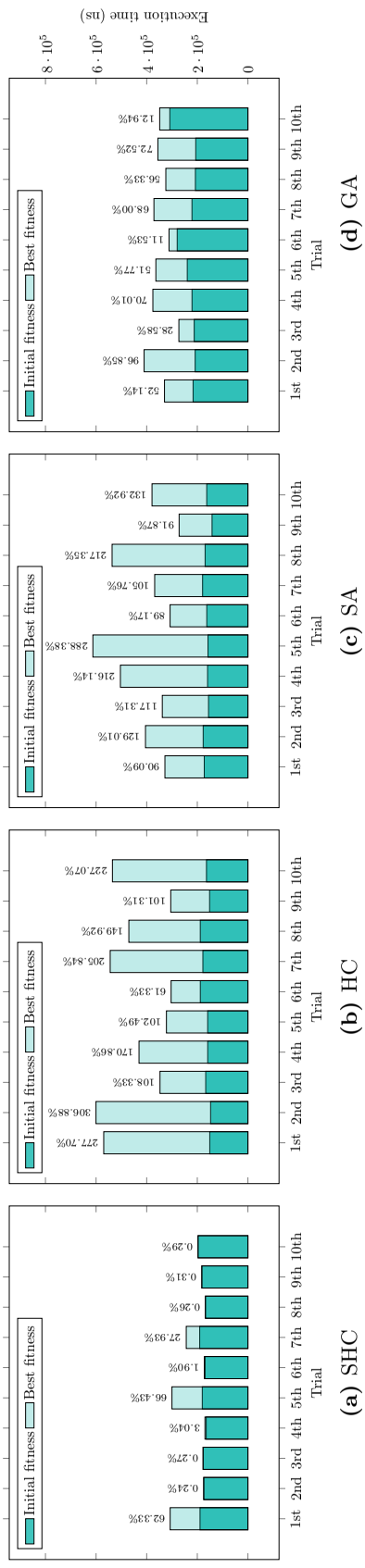
(a) SHC **(b)** HC **(c)** SA **(d)** GA

Figure 3.13: Results of 10 trials obtained by each metaheuristic algorithm to the sextic equation



(a) SHC (b) HC (c) SA (d) GA

Figure 3.14: Results of 10 trials obtained by each metaheuristic algorithm to the octic equation



(a) SHC (b) HC (c) SA (d) GA

Figure 3.15: Results of 10 trials obtained by each metaheuristic algorithm to the decic equation

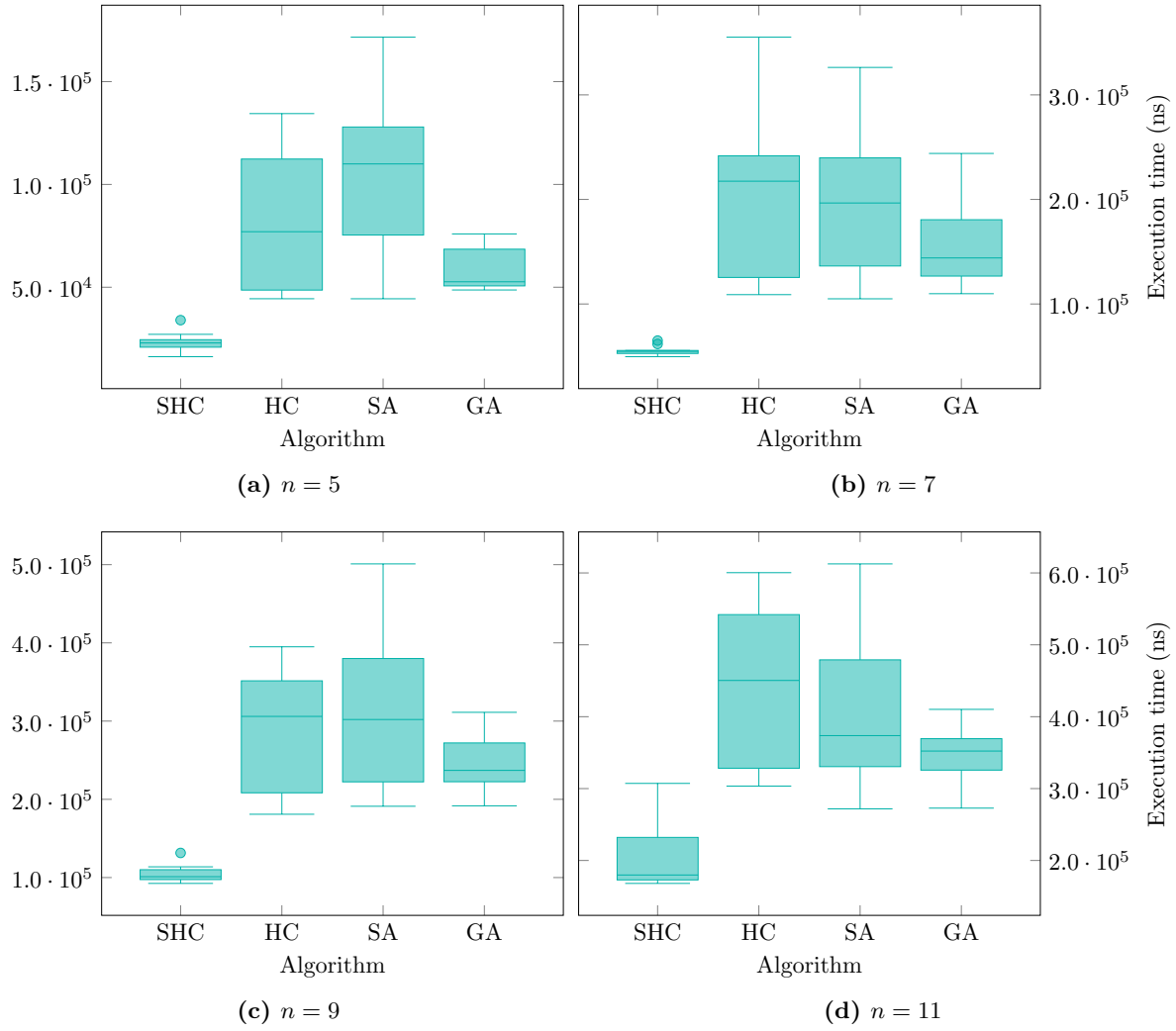
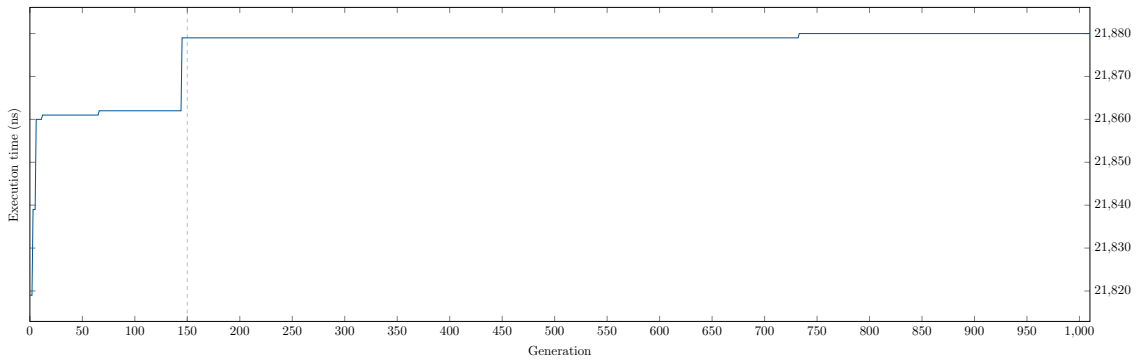


Figure 3.16: Distribution of the best fitness values of 10 trials of each metaheuristic algorithm on the polynomial equation formed $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$

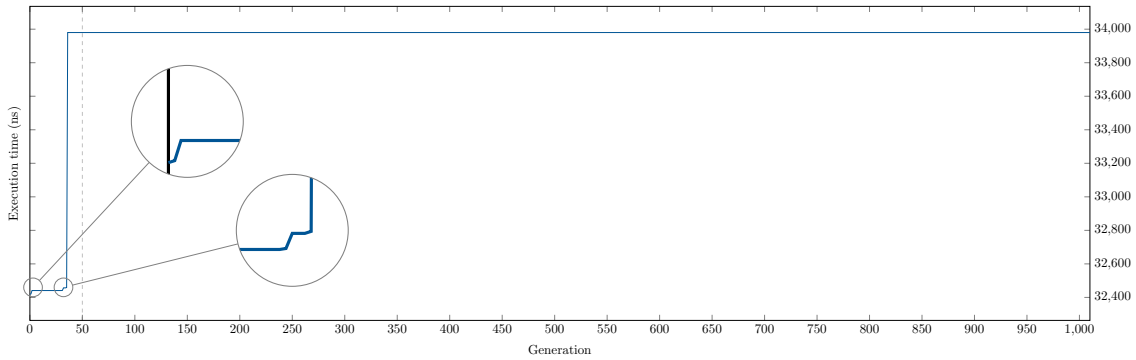
Table 3.5: Summary of the highest fitness value of 10 trials of each metaheuristic algorithm over different input arguments for the polynomial solver

	Arguments			
Algorithm	**5**	**7**	**9**	**11**
SHC	33,980*	64,984*	131,467*	307,227*
HC	134,449	355,198†	394,979	600,362
SA	171,629.5†	326,249	500,971†	612,587.5†
GA	75,917	243,980	311,219	410,202
RS	66,978	204,990	317,108	558,421.5

* The worst performer. † The best performer.



(a) 1st trial



(b) 5th trial

Figure 3.17: Results obtained by SHC to the quartic equation in the 1st and 5th trials

Both of the line graphs in Figure 3.17 reveal evidence that the global best fitness value improved at the early stage of the search for a while and then it stopped improving. It is due to the overly-restrictive neighbourhood that does not allow any improvements to be found one hop away. Although SHC allows a big jump to a new area when there is no improvement in its current neighbourhood (as indicated in line 4 of Algorithm 3), it still has less chance of escaping from a local optimum in comparison with HC, where it is more flexible to jump to anywhere over the space, at the same number of evaluations.

Furthermore, as illustrated in Figures 3.12 to 3.15, GA regularly started its initial fitness with a higher value compared with other techniques. The reason is that it is a population-based approach; therefore, it gets more chance to select the best candidate solution from its population.

On the other hand, the two single-solution based metaheuristics, i.e. HC and SA, usually began the search process with a low fitness value, which is similar to SHC, but both of them delivered more desirable fitness value at the end, as depicted in Figures 3.12 to 3.15.

In addition, Figure 3.16 shows that, in almost all cases, the final fitness values among ten trials of SHC were very low and close to each other, except for the problem input size of 11, where there were slightly varied between the trials, and the extreme values appeared. On the contrary, the distributions of trials' best fitness from HC and SA fluctuated wildly in all cases, but overall they performed very well as displayed by the whiskers on seeking values of the coefficients that heighten the routine's execution time. Although GA was largely capable of stressing the polynomial solver and gave better results than SHC, it was inferior to HC and SA in all the cases.

Research Question 2 The second research question is answered by comparing the metaheuristic techniques with RS. We executed RS for the same number of evaluations at 10,100. As shown in Table 3.5, the longest execution time among ten trials of RS was worse than the ones of SA and HC in all the cases, but better than the ones of GA in problem input sizes of 9 and 11. SHC was the worst in all cases to seek values of coefficients that maximise the polynomial solver's execution time. SA substantially outperformed other techniques, excepting the problem input size of 7, where HC was superior.

Regarding the best fitness values (the ones marked by †) shown in Table 3.5, the coefficients that produced such extreme execution times of the polynomial solver, as well as the number of iterations required for each of them, are listed in Table 3.6.

As shown in Table 3.6, these coefficient vectors maximised the number of QR iterations for convergence as earlier described in Section 3.3.1. For instance, in the quartic equation, the initial vector (i.e. 30,166; 15,894; 23,950; −15,881 and −2,509) took 17,960.0 ns to execute the polynomial root-finding function as only six QR iterations is needed. The best vector shown in Table 3.6, on the other hand, took 171,629.5 ns since it repeated the use of QR refactorisation 59 times.

Table 3.6: Best values of coefficients (Chapter 3)

Arguments	Coefficient vector	Iterations
5	21,508; 15,894; −22,267; −20,437 and 17,985	9
7	3,797; −24,011; 28,203; 26,154; −13,873; −6,095 and 2,622	71
9	26,880; 2,291; 23,227; 8,105; −12,774; 4,007; −29,306; −8,575 and 20,905	72
11	−15,999; 18,598; 1,673; −23,003; −29,935; −10,917; 6,219; 6,246; 19,083; 8,416 and −16,062	52

Another example is the decic equation, where the initial coefficient vector (i.e. 9,689; 18,598; 3,954; −23,003; −29,935; −444; −24,838; −30,125; 22,755; 8,416 and −16,062) only causes 21 iterations to execute the function for 157,728 ns, whereas the best vector presented in Table 3.6 requires 52 iterations, which took 612,587.5 ns. Besides, the problem size also significantly impacts the execution time of the root-finding function as more operations are invoked.

The approximate roots of polynomials, where their coefficients maximised the execution times of GSL's polynomial root-finder function, are summarised in Tables B.1 to B.4 of Appendix B.

We further validated these best input arguments by rerunning them with the GSL's polynomial solver. In particular, each best coefficient vector was exclusively run on five different COMX-P4080 boards 100 times. The aim of this validation is to show the variation in speed among the development boards. The execution times obtained by each P4080 board are plotted in Figure 3.18.

Specifically, each box plot presents the distribution of 100 execution times obtained by running each coefficient vector with the polynomial root-finder on each development board. A (red) line crossing all boxes of each diagram indicates the actual best fitness value gained from each experiment in this section.

According to Figure 3.18, there was almost no variation of execution times on each development board in all the cases, as revealed by the Interquartile Range (IQR), which was nearly close to zero, albeit only a few extremal values were found. Such midspreads of all development boards were also nearly the same in each case. Furthermore, the actual best fitness values are within the ranges of distributions over five different boards. In the case of octic equation ($n = 9$), although the actual value was outside the midspread and slightly higher than the upper quartile, the number of QR iterations achieved by the actual experiment and this validation was identical at 72.

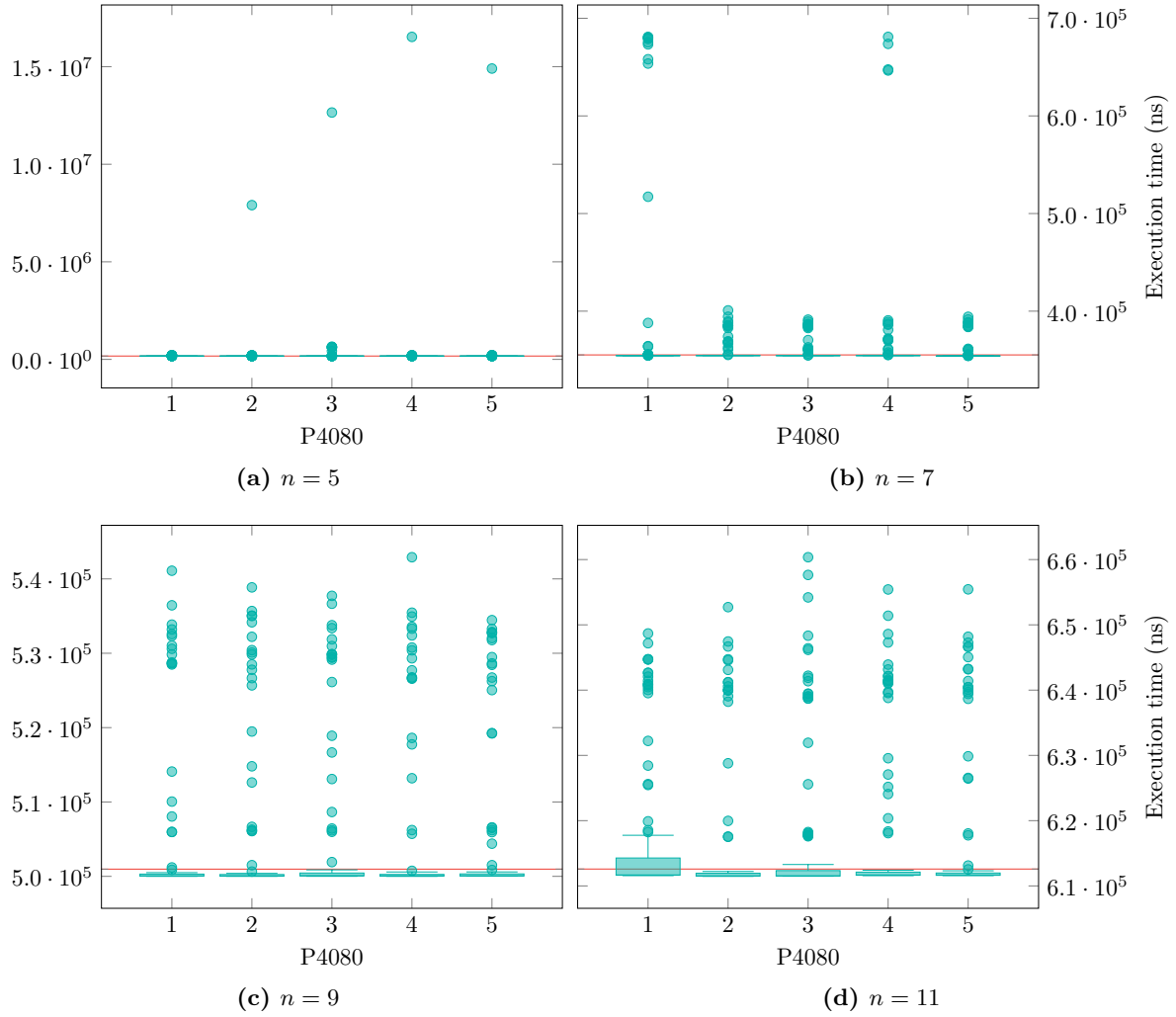


Figure 3.18: Distribution of 100 execution times of polynomial solver with best input arguments obtained from 5 P4080 boards (Chapter 3)

3.6 Experiment II—Multi-Threaded Routines

3.6.1 Objectives

Ideally, a parallel program has great potential to get benefit from multicore processing. However, as previously discussed in Section 2.1.4, interference may occur on this shared component environment during execution and can affect the timing constraints of the system. In this experiment, we apply metaheuristic algorithms to find temporal test inputs that may trigger a multi-threaded routine, i.e. a sorting algorithm, running on the COMX-P4080 development board, to violate performance timing requirements. Hence, the problem statement in this experiment is:

Problem Statement: *We seek sequences of values that maximise the execution time of the sorting.*

Accordingly, the research questions addressed are:

Research Question 1: *Among the metaheuristics, which technique is the best for seeking sequences of values that maximise the execution time of the sorting?*

Research Question 2: *Are metaheuristics effective in seeking sequences of values that maximise the execution time of the sorting?*

Research Question 3: *Does the number of threads affect the execution time of the sorting?*

3.6.2 Preparation

As mentioned earlier in Section 3.5.2, this experiment also focused on the `int` type with its range from −32,768 to 32,767. Each sorting algorithm was executed with two different numbers of input arguments, i.e. 100 and 200, respectively. We decided to expand the number of input arguments to the hundreds since it is adequate enough—not too small, not too big—to be executed by a parallel sorting routine. The domain cardinality of each size of integer inputs is summarised in Table 3.7.

Table 3.7: Domain cardinalities of the input arguments under consideration for the sorting routines

Data type	Arguments	Input range	Cardinality
`int`	100	−32,768 to 32,767	$65{,}536^{100}$
`int`	200	−32,768 to 32,767	$65{,}536^{200}$

The algorithm parameter settings used in this empirical experiment are also the same as in Experiment I (Section 3.5), as listed in Table 3.2. Again, for SHC, the number of generations and λ depend on the test input size. The parameter setting of SHC in this experiment is illustrated in Table 3.8.

3.6.3 Method

The methodology in this experiment is quite similar to the previous experiment described in Section 3.5.3. It differs only in that we additionally aimed to investigate the effect of threads on the execution time of the multi-threaded sorting routines as stated in the last research question of this experiment. Therefore, each metaheuristic algorithm was executed with each problem input size of each sorting function and with a given number of threads for ten trials. The initial seeds, which are also procured from `random.org`, for bubble sort, shell sort, quicksort and merge sort are summarised in Tables 3.9 to 3.11, respectively.

Table 3.8: Parameter settings for SHC (Experiment II)

	Arguments	
Parameter	**100**	**200**
Generations	52	27
Population size	2	2
μ	1	1
λ	200	400
δ	50	50
P_m	0.1	0.1
Evaluations	10,202	10,402

Table 3.9: Initial seeds of metaheuristic algorithms for different input arguments of bubble sort

(a) 100

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	25,049	−159	−19,509
HC	−5,315	−30,808	−914
SA	−6,540	−20,893	31,800
GA	−1,046	−3,028	−31,600
RS	9,175	29,673	−24,378

(b) 200

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	15,764	−4,326	7,920
HC	−13,317	2,882	28,075
SA	−28,268	11,296	15,458
GA	−16,048	−3,199	5,809
RS	4,364	−5,413	−13,410

Table 3.10: Initial seeds of metaheuristic algorithms for different input arguments of shell sort

(a) 100

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	−1,819	−30,387	31,768
HC	−7,954	31,167	−19,256
SA	12,504	3,051	16,987
GA	12,119	−32,102	−5,037
RS	−16,149	11,739	−27,807

(b) 200

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	−2,742	−24,005	17,706
HC	−29,459	15,036	9,706
SA	−29,760	−20,501	32,220
GA	−10,193	−6,743	14,881
RS	24,223	−28,015	−13,788

Table 3.11: Initial seeds of metaheuristic algorithms for different input arguments of quicksort and merge sort

(a) Quicksort

	Arguments	
Algorithm	**100**	**200**
SHC	24,499	−14,056
HC	−24,723	−11,419
SA	−2,782	25,511
GA	−24,139	32,463
RS	6,650	−30,041

(b) Merge sort

	Arguments	
Algorithm	**100**	**200**
SHC	4,541	8,372
HC	28,358	−17,393
SA	−8,228	5,033
GA	−30,053	6,812
RS	19,193	15,532

3.6.4 Results

The results in this experiment are presented in the same way as in the preceding experiment. In particular, for each sorting algorithm, the bar charts illustrate the differences between the initial and final fitness values achieved from ten trials of each metaheuristic search on stressing it with a specific number of threads and input arguments, i.e. 100 and 200, respectively. As previously described in Section 3.3.2, for bubble sort and shell sort, the number of threads was specified to be 1 to 3, whereas the thread numbers cannot be assigned to quicksort and merge sort.

After that, the box-and-whisker plots are used to exhibit distributions among the trials' best fitness values of each metaheuristic method over different numbers of threads and input arguments. Finally, the longest execution time amongst trials of each technique is summarised in the tables.

For ease of reference, Table 3.12 below contains a summary of figures and tables, which are relevant to the results of each multi-threaded sorting routine. (For an electronic copy, the reader may use the hyperlinks provided within the table to simply access to a particular figure or table.)

Table 3.12: Summary of figures and tables related to the results of each sorting routine

Routine	**Bar charts**	**Box plots**	**Tables**
Bubble sort	Figures 3.19 and 3.20	Figure 3.21	Table 3.13
Shell sort	Figures 3.22 and 3.23	Figure 3.24	Table 3.14
Quicksort	Figures 3.25 and 3.26	Figure 3.27	Table 3.15
Merge sort	Figures 3.28 and 3.29	Figure 3.30	Table 3.16

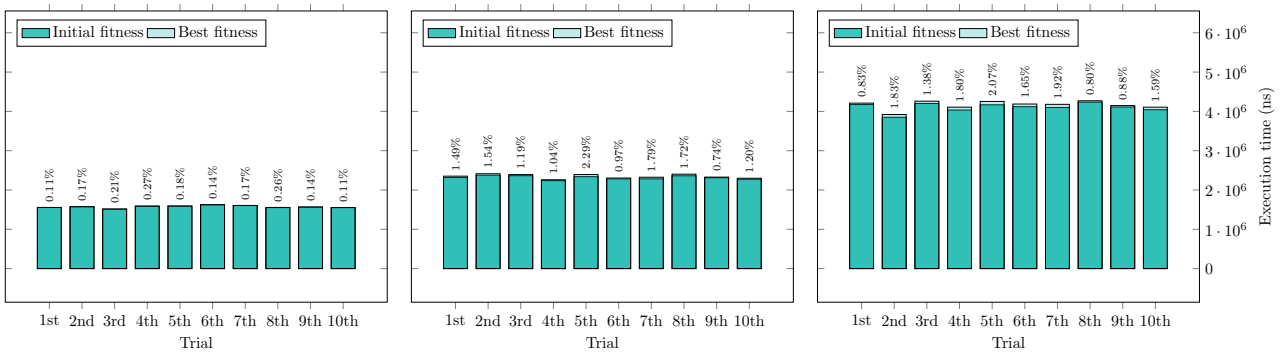
(a) SHC—1 thread **(b)** SHC—2 threads **(c)** SHC—3 threads

(d) HC—1 thread **(e)** HC—2 threads **(f)** HC—3 threads

(g) SA—1 thread **(h)** SA—2 threads **(i)** SA—3 threads

(j) GA—1 thread **(k)** GA—2 threads **(l)** GA—3 threads

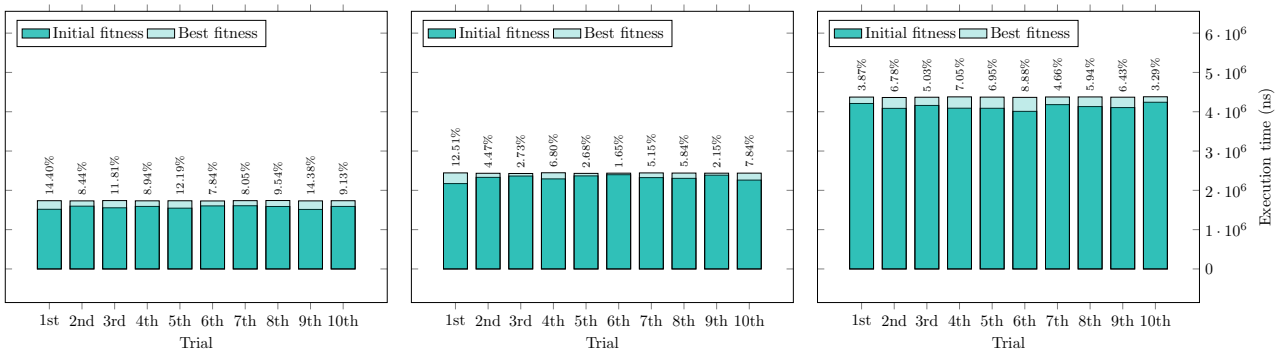
Figure 3.19: Results of 10 trials obtained by each metaheuristic algorithm to bubble sort with 100 input arguments



(a) SHC—1 thread

(b) SHC—2 threads

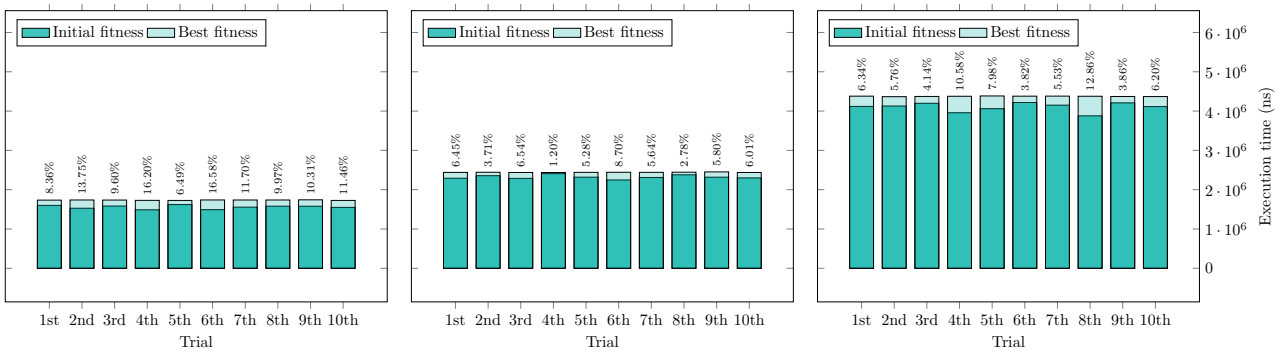
(c) SHC—3 threads



(d) HC—1 thread

(e) HC—2 threads

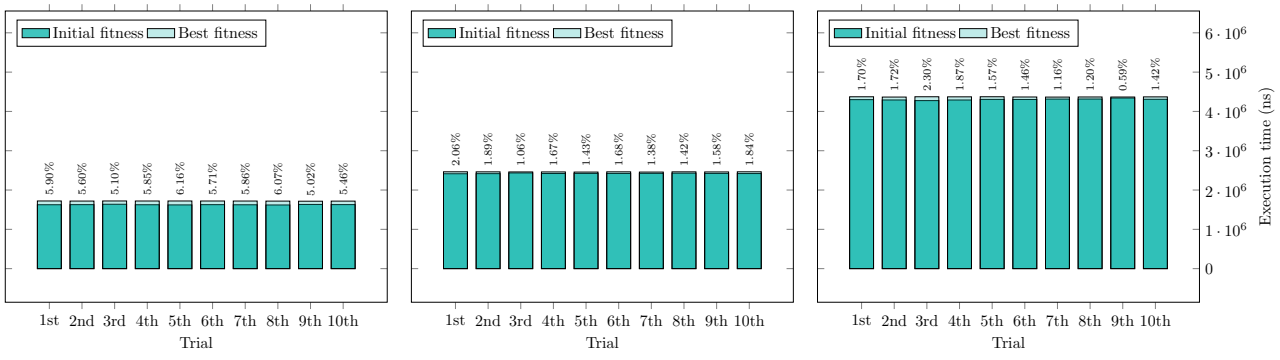
(f) HC—3 threads



(g) SA—1 thread

(h) SA—2 threads

(i) SA—3 threads

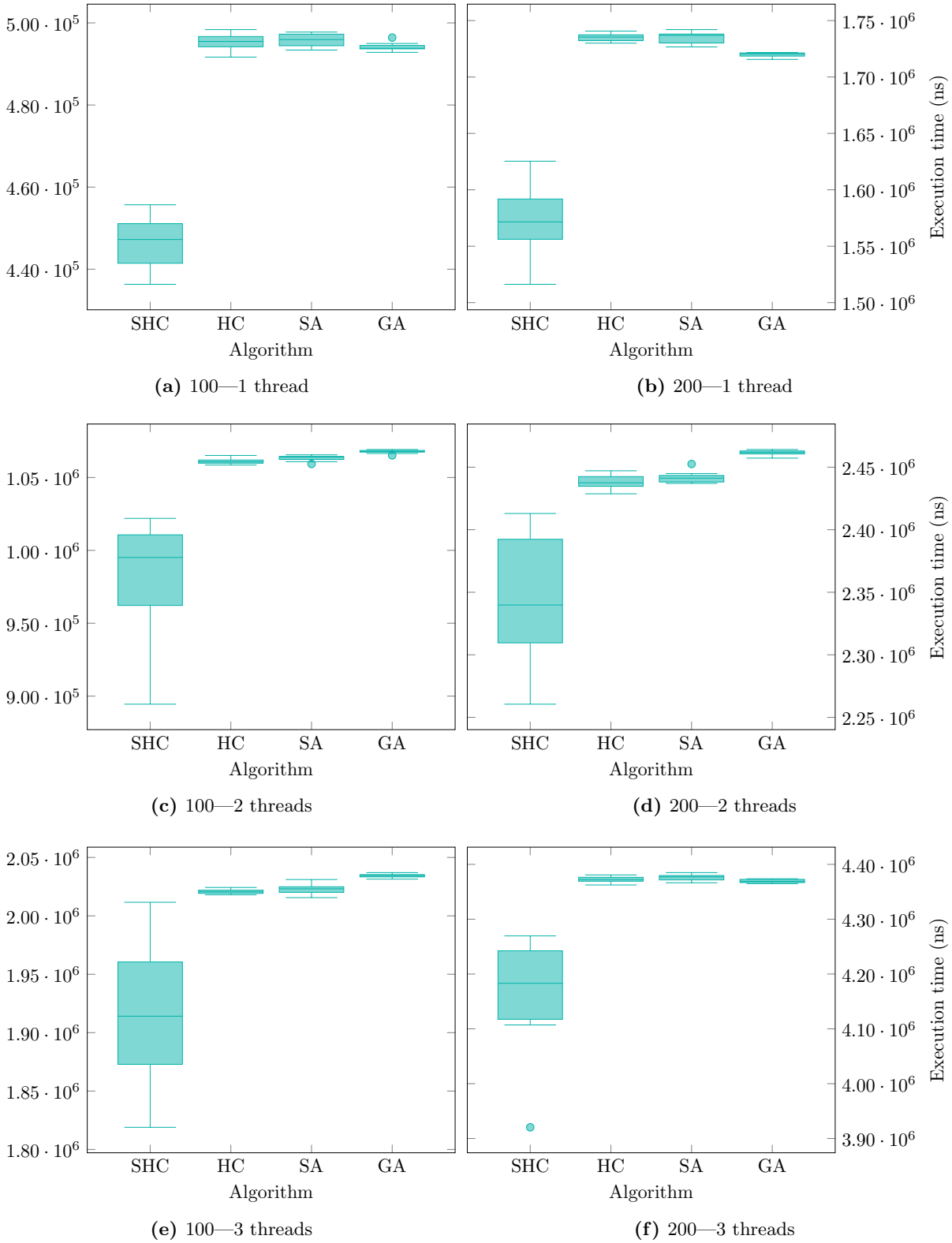


(j) GA—1 thread

(k) GA—2 threads

(l) GA—3 threads

Figure 3.20: Results of 10 trials obtained by each metaheuristic algorithm to bubble sort with 200 input arguments



(a) 100—1 thread **(b)** 200—1 thread

(c) 100—2 threads **(d)** 200—2 threads

(e) 100—3 threads **(f)** 200—3 threads

Figure 3.21: Distribution of the best fitness values of 10 trials of each metaheuristic algorithm on bubble sort with 100 and 200 input arguments

Table 3.13: Summary of the highest fitness value of 10 trials of each metaheuristic algorithm over the different number of threads of bubble sort

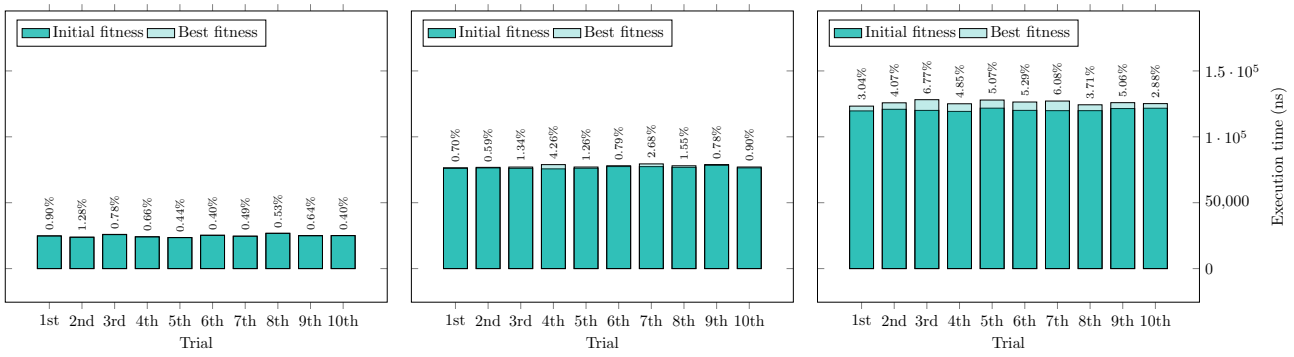
(a) 100

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	455,695.5*	1,021,941.5*	2,011,617*
HC	498,359.5†	1,065,114	2,024,320
SA	497,770.5	1,065,540.5	2,031,089.5
GA	496,420.5	1,069,218.5†	2,037,010†
RS	474,795.5	1,057,870	2,015,250

(b) 200

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	1,625,372.5*	2,412,950*	4,269,500*
HC	1,740,700	2,447,100	4,380,680
SA	1,741,974.5†	2,452,520.5	4,384,920†
GA	1,721,700.5	2,464,131†	4,373,701.5
RS	1,649,699	2,431,220	4,346,010

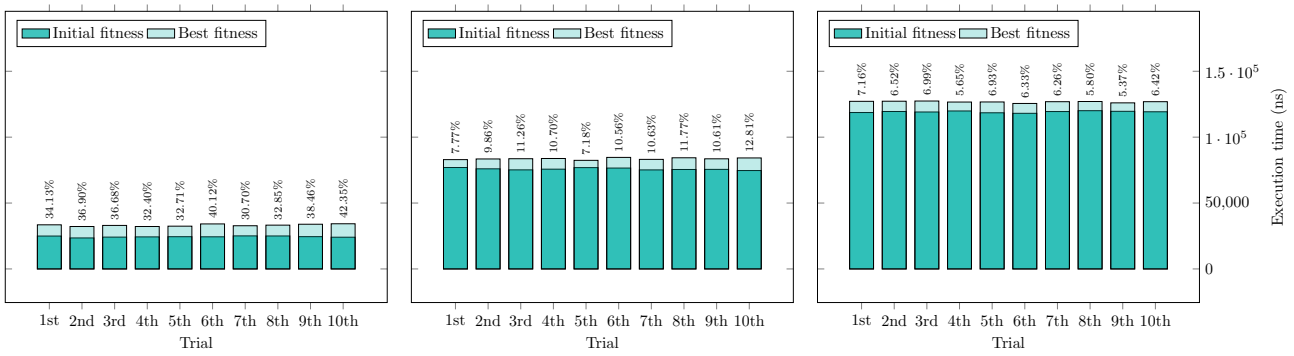
* The worst performer. † The best performer.



(a) SHC—1 thread

(b) SHC—2 threads

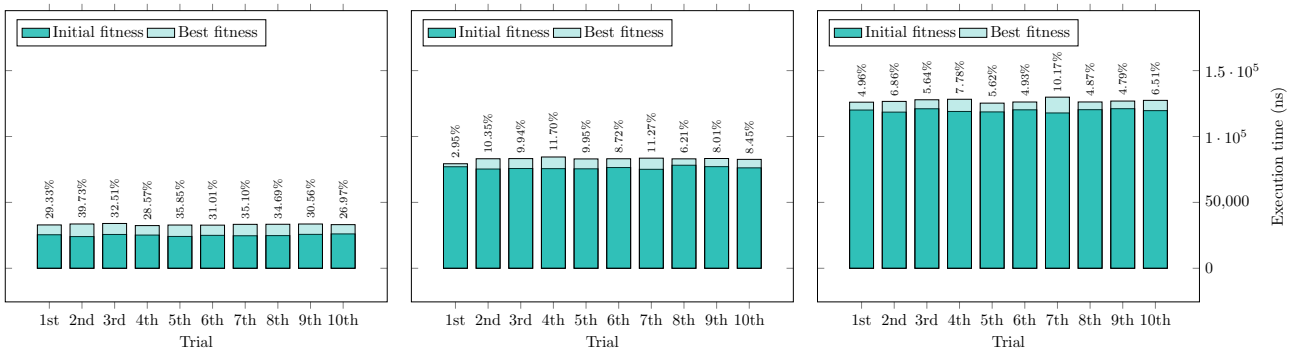
(c) SHC—3 threads



(d) HC—1 thread

(e) HC—2 threads

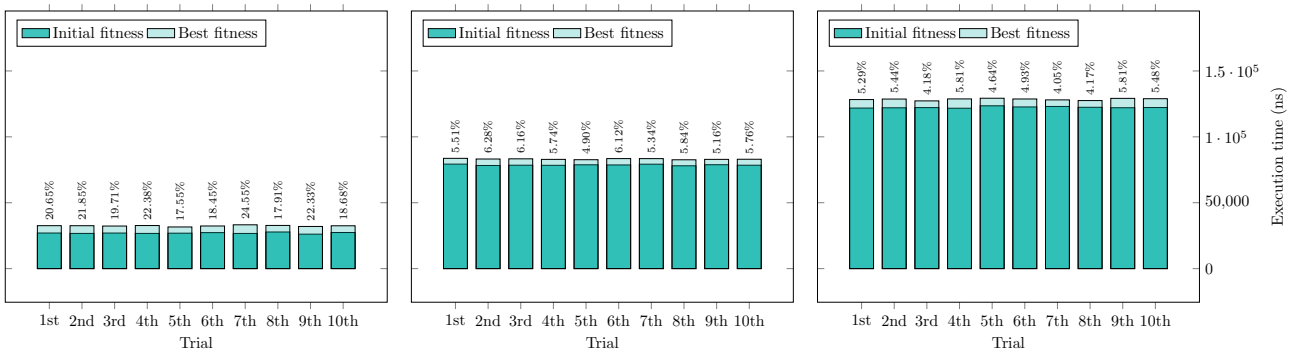
(f) HC—3 threads



(g) SA—1 thread

(h) SA—2 threads

(i) SA—3 threads

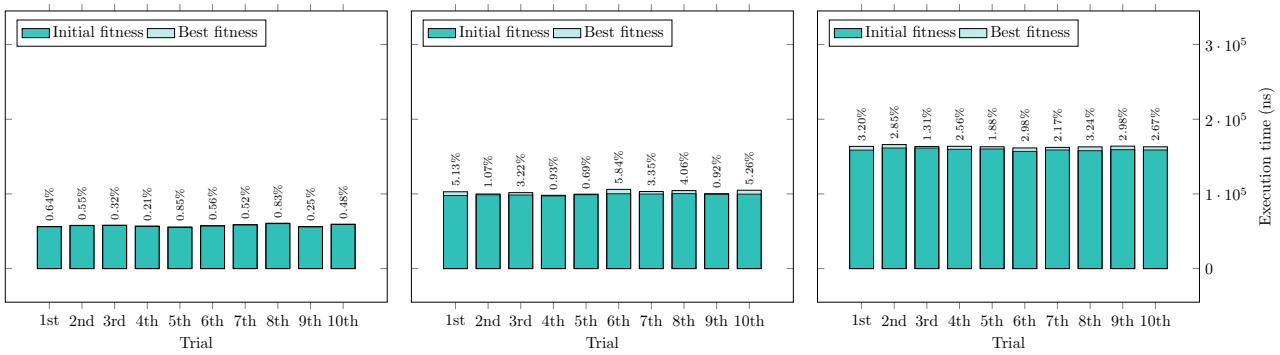


(j) GA—1 thread

(k) GA—2 threads

(l) GA—3 threads

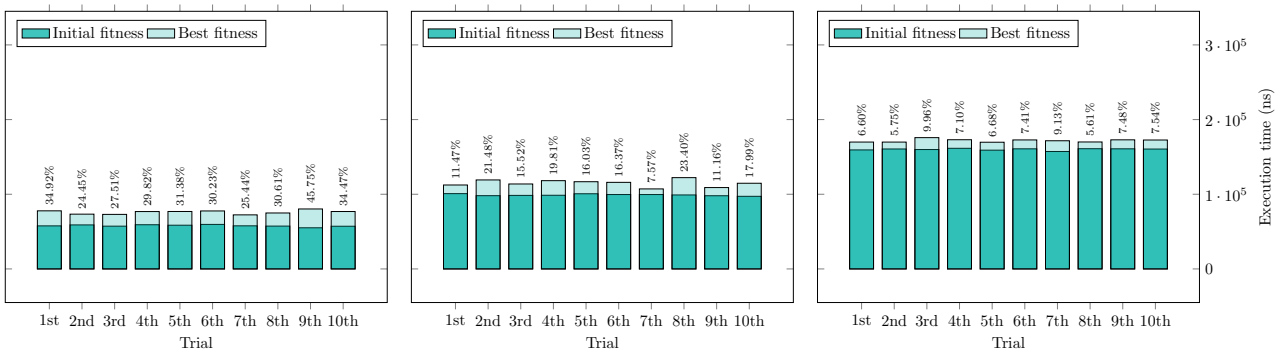
Figure 3.22: Results of 10 trials obtained by each metaheuristic algorithm to shell sort with 100 input arguments



(a) SHC—1 thread

(b) SHC—2 threads

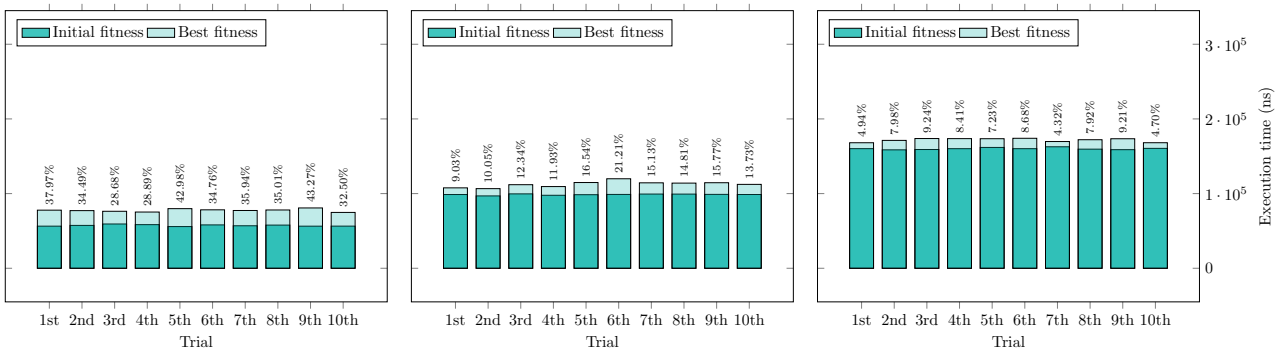
(c) SHC—3 threads



(d) HC—1 thread

(e) HC—2 threads

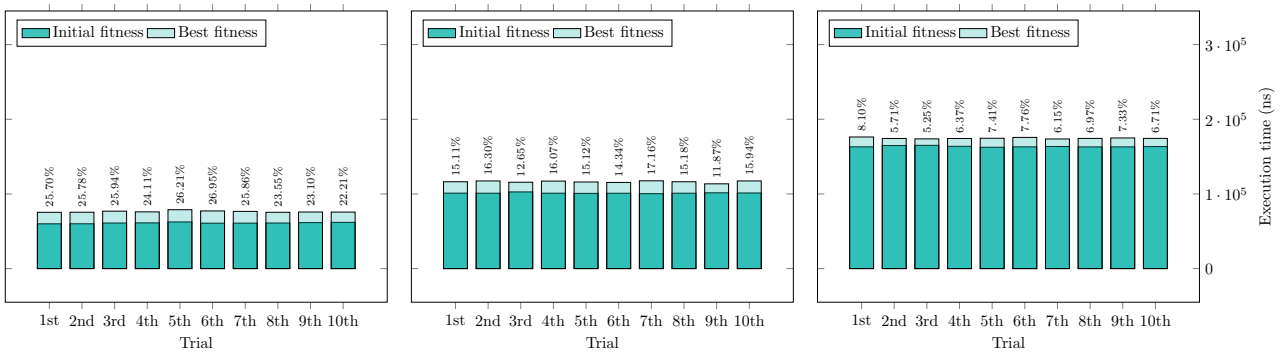
(f) HC—3 threads



(g) SA—1 thread

(h) SA—2 threads

(i) SA—3 threads

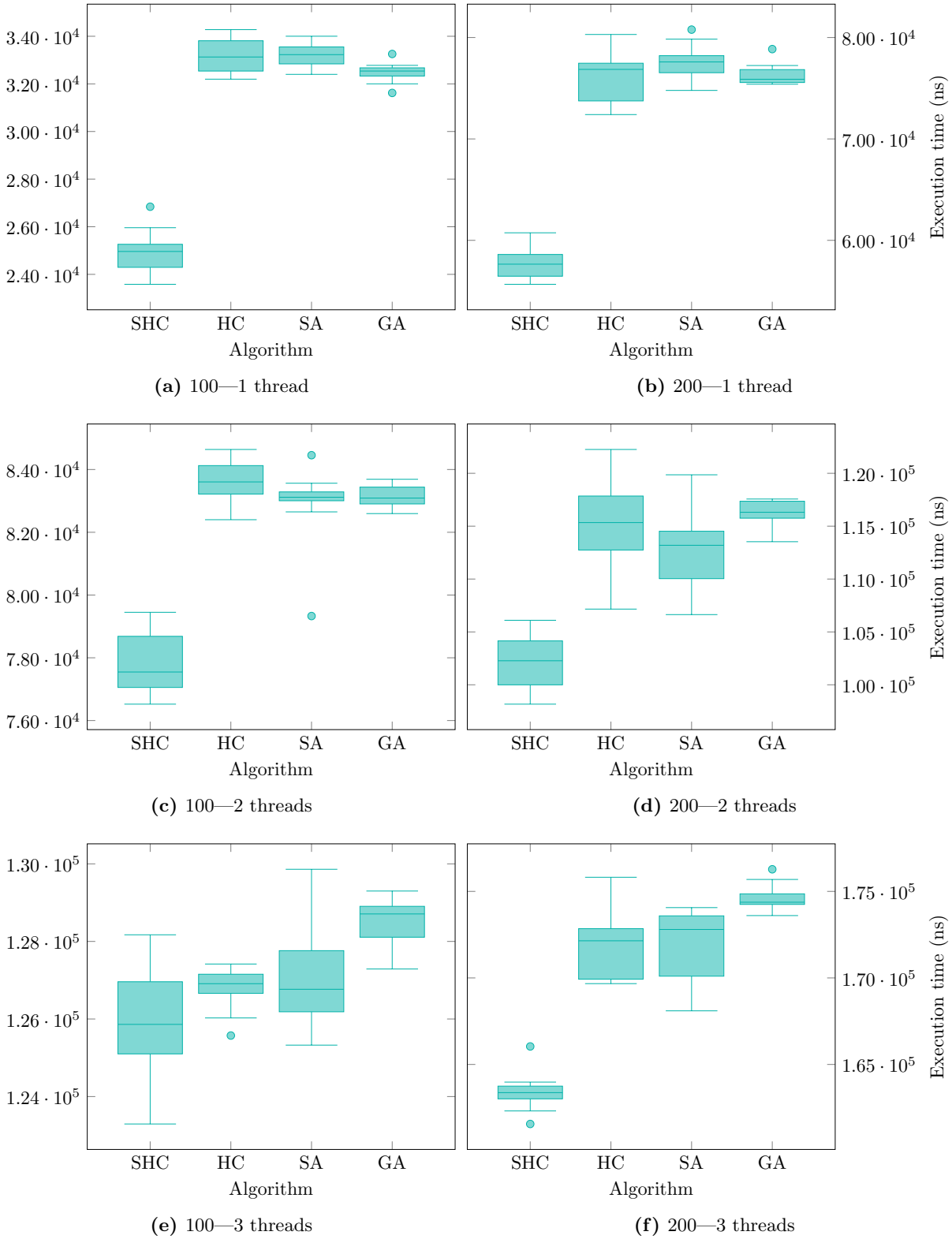


(j) GA—1 thread

(k) GA—2 threads

(l) GA—3 threads

Figure 3.23: Results of 10 trials obtained by each metaheuristic algorithm to shell sort with 200 input arguments



(a) 100—1 thread **(b)** 200—1 thread

(c) 100—2 threads **(d)** 200—2 threads

(e) 100—3 threads **(f)** 200—3 threads

Figure 3.24: Distribution of the best fitness values of 10 trials of each metaheuristic algorithm on shell sort with 100 and 200 input arguments

Table 3.14: Summary of the highest fitness value of 10 trials of each metaheuristic algorithm over the different number of threads of shell sort

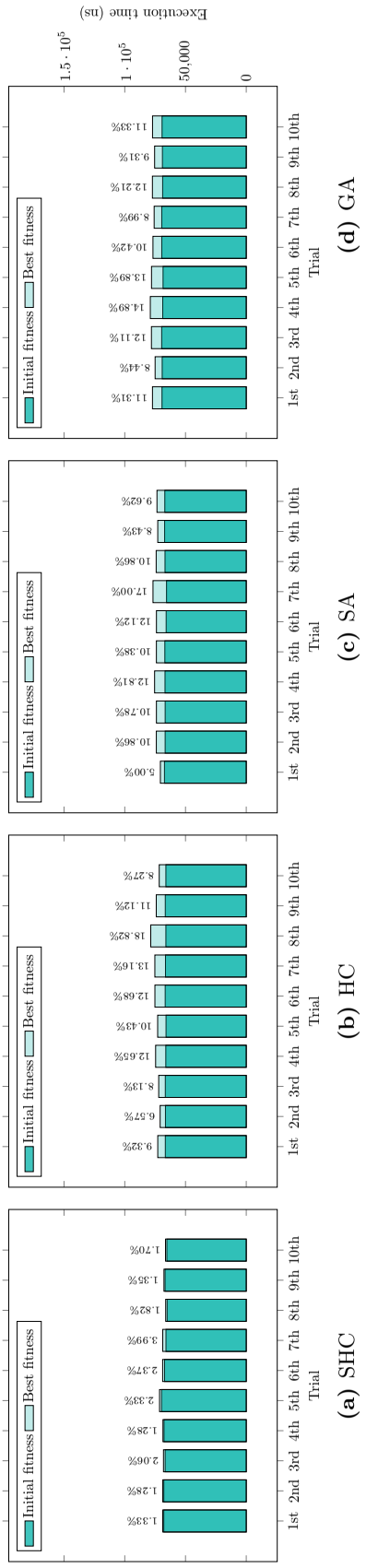
(a) 100

	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	26,841*	79,450*	128,172
HC	34,279†	84,639†	127,417.5
SA	34,002	84,455.5	129,860†
GA	33,256	83,689	129,299.5
RS	28,481	79,988	123,396.5*

(b) 200

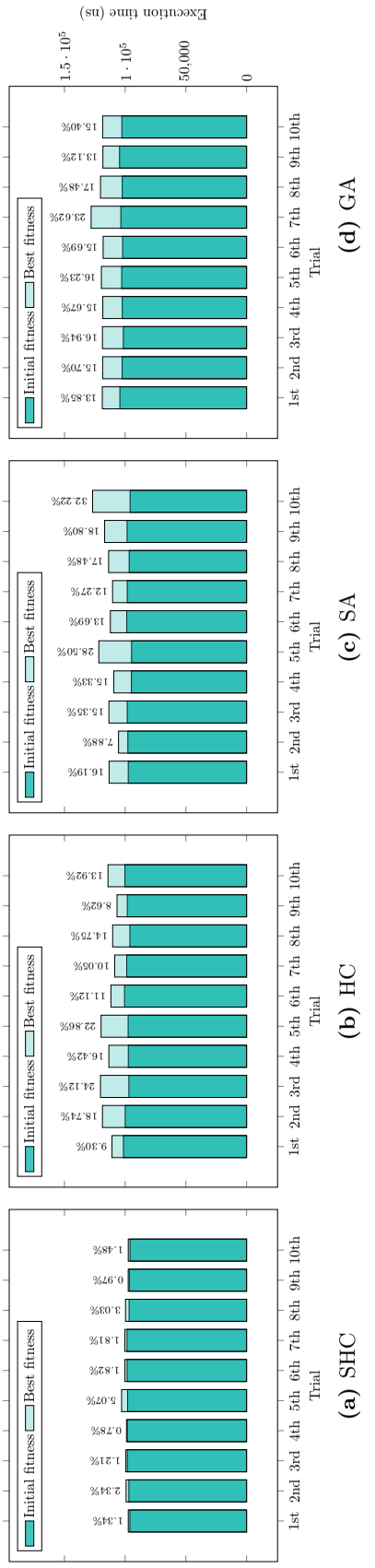
	Thread(s)		
Algorithm	**1**	**2**	**3**
SHC	60,743*	106,100	166,040*
HC	80,300	122,254.5†	175,816
SA	80,779†	119,858	174,062.5
GA	78,858	117,566	176,276.5†
RS	64,197	105,420*	166,087.5

* The worst performer. † The best performer.



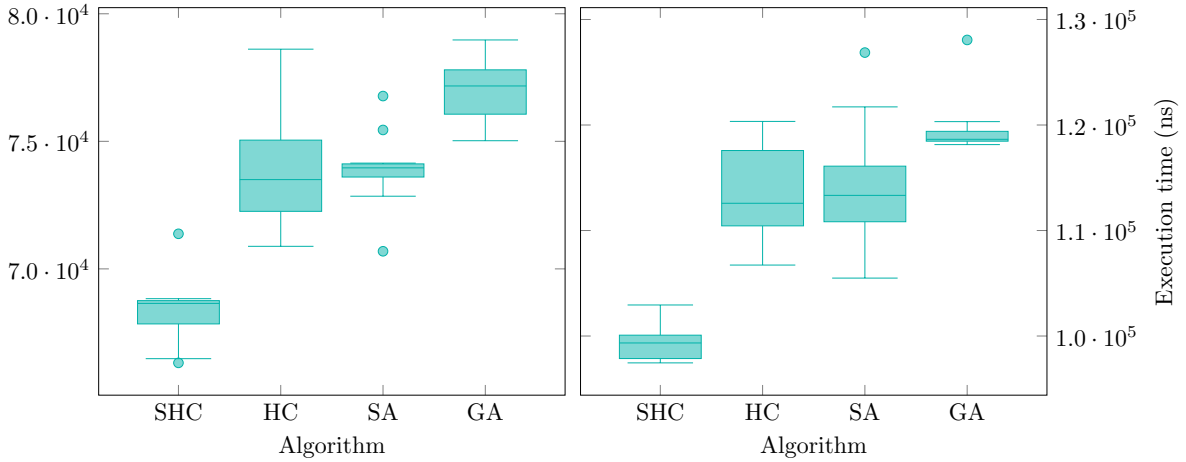
(a) SHC (b) HC (c) SA (d) GA

Figure 3.25: Results of 10 trials obtained by each metaheuristic algorithm to quicksort with 100 input arguments



(a) SHC (b) HC (c) SA (d) GA

Figure 3.26: Results of 10 trials obtained by each metaheuristic algorithm to quicksort with 200 input arguments



(a) 100 inputs **(b)** 200 inputs

Figure 3.27: Distribution of the best fitness values of 10 trials of each metaheuristic algorithm on quicksort with 100 and 200 input arguments

Table 3.15: Summary of the highest fitness value of 10 trials of each metaheuristic algorithm on quicksort

	Arguments	
Algorithm	**100**	**200**
SHC	71,375	102,942.5*
HC	78,612.5	120,340.5
SA	76,774.5	126,864.5
GA	78,974†	128,062†
RS	70,744*	106,664.5

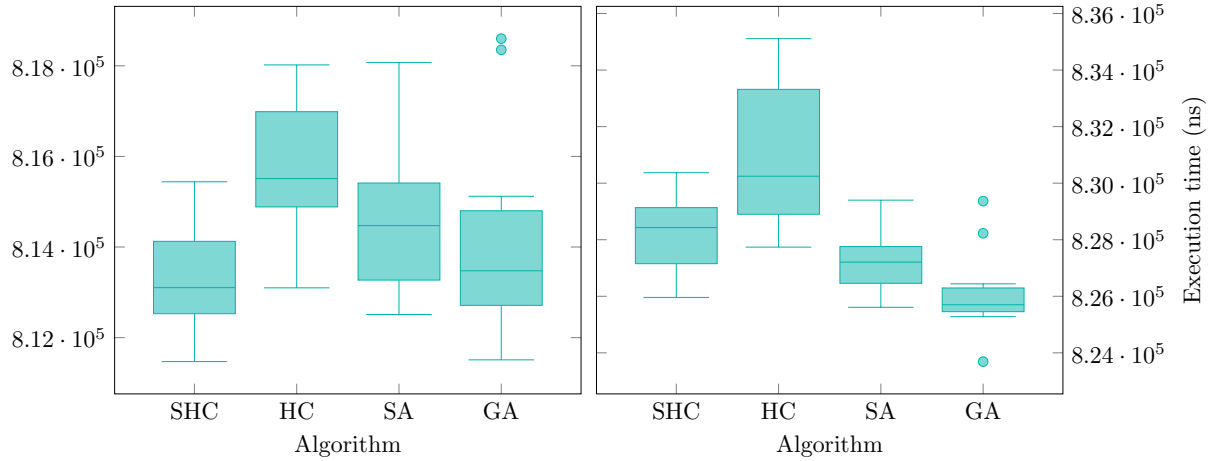
* The worst performer. † The best performer.

(a) SHC **(b)** HC **(c)** SA **(d)** GA

Figure 3.28: Results of 10 trials obtained by each metaheuristic algorithm to merge sort with 100 input arguments

(a) SHC **(b)** HC **(c)** SA **(d)** GA

Figure 3.29: Results of 10 trials obtained by each metaheuristic algorithm to merge sort with 200 input arguments



(a) 100 inputs **(b)** 200 inputs

Figure 3.30: Distribution of the best fitness values of 10 trials of each metaheuristic algorithm on merge sort with 100 and 200 input arguments

Table 3.16: Summary of the highest fitness value of 10 trials of each metaheuristic algorithm on quicksort

	Arguments	
Algorithm	**100**	**200**
SHC	815,440.5*	830,370.5
HC	818,020	835,110†
SA	818,071	826,947*
GA	818,599†	829,366.5
RS	818,444.5	827,390

* The worst performer. † The best performer.

3.6.5 Discussion and Conclusions

The first two research questions are answered through a comparison of the effectiveness among the metaheuristics themselves, and a comparison of the effectiveness between the metaheuristics and RS, respectively.

Research Question 1 The first research question is discussed below:

With regard to all the bar charts (presented in Figures 3.19, 3.20, 3.22, 3.23, 3.25, 3.26, 3.28 and 3.29), all single-point metaheuristic algorithms, i.e. SHC, HC and SA, began with roughly the same performance on seeking sequences of values that maximise the execution time of the sortings, whereas the temporal test inputs sought by a population-based metaheuristic algorithm, i.e. GA, slightly caused a higher execution time. However, at the end of the search process on the bubble sort, shell sort and quicksort, all of these metaheuristic approaches gave likely the same improved execution times, except for SHC that was trapped on local optima. Surprisingly, SHC broadly outperformed GA in the case of merge sort as shown in Figures 3.28 and 3.29.

Additionally, in accordance with the box-and-whisker plots (depicted in Figures 3.21, 3.24, 3.27 and 3.30), the ability of each metaheuristic algorithm to search for the sequences of values that maximise the execution time was divergent among the sorting routines.

In particular, among ten trials of each algorithm applied on the bubble sort, the final fitness values of HC, SA and GA were approximately constant over different numbers of threads, while the SHC's fitness values were not only varying but also inefficient as illustrated in Figure 3.21.

In cases of shell sort (presented in Figure 3.24) and quicksort (presented in Figure 3.27), HC, SA and GA performed almost the same as in bubble sort in terms of stability. Yet, there were some wide variations among trials in some cases, such as SA with the problem size of 100 and 3 threads in shell sort (Figure 3.24e), HC with the problem size of 200 and 2 threads in shell sort (Figure 3.24d) and HC with both input sizes in quicksort (Figures 3.27a and 3.27b).

Compared with the other three approaches, SHC performed worse on seeking the temporal test inputs in both bubble sort and shell sort, but it was exceptionally better than SA and GA on average in the case of merge sort with the input arguments of 200 as shown in Figure 3.30b. It was also able to seek the sequences of 100 values that escalate the execution time of merge sort to be higher than those of GA as displayed in Figure 3.30a.

Research Question 2 The second research question is discussed below:

In this experiment, we also executed RS for ten trials with the same number of evaluations at 10,100 in order to compare its results with the metaheuristic methods. Accordingly, in Tables 3.13 to 3.16, either HC, SA or GA were predominantly capable of seeking temporal test inputs that produce the longest execution time. Broadly, SHC and RS fell behind that of the other techniques. For example, in bubble sort, SHC was the worst and followed by RS in all cases as presented in Table 3.13. For merge sort, however, RS surpassed HC and SA on the problem input size of 100, and SA was the worst on the problem input size of 200 as shown in Table 3.16.

Overall, there was not much difference among these four metaheuristics in the ability to search for temporal test inputs that maximise the execution time of the sorting due to the large search space sizes of both problem input sizes as formerly summarised in Table 3.7.

The input arguments that led to the best fitness values (the ones marked by † in Tables 3.13 to 3.16) are listed in Tables D.1 to D.3 of Appendix D. As summarised in Table 3.17, those best input arguments require the higher numbers of swaps/comparisons than the initial test inputs. As a result, they took longer execution times for sorting.

Additionally, as previously described in Section 3.3.2, the worst-case permutation of a particular sorting has its specific pattern or characteristic of the input sequence. Accordingly, we plotted line graphs to illustrate the patterns of the best input sequences as shown in Figures 3.31 to 3.35. Explicit examples of the worst-case permutations are shown in Figures 3.31a and 3.32a, where the sequences of the input arguments are nearly in the reverse order, corresponding to the worst-case scenario of bubble sort.

Similar to Experiment I, we also validated these best sequences of arrays by rerunning them with the parallel sortings in this experiment. Figures 3.36 to 3.38 depict distributions of 100 execution times obtained by running the best sequences of arrays with sorting routines on five P4080 development boards.

Table 3.17: Summary of the numbers of swaps/comparisons of the best sequences of values for sortings compared to the initial sequences

(a) Bubble sort (number of swaps)

Arguments	Thread(s)	Initial sequence	Best sequence
100	1	4,345	6,319
	2	4,072	4,455
	3	4,276	4,974
200	1	14,658	23,626
	2	15,496	19,314
	3	15,915	16,535

(b) Shell sort (number of swaps)

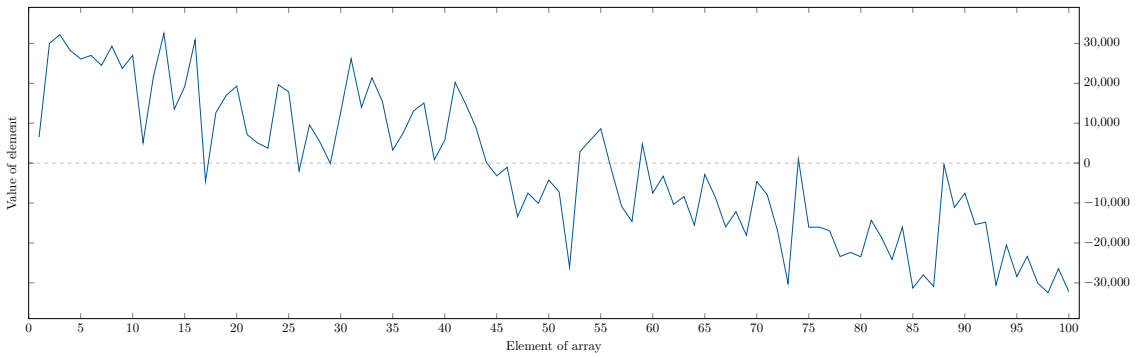
Arguments	Thread(s)	Initial sequence	Best sequence
100	1	501	972
	2	527	887
	3	467	638
200	1	1,134	2,280
	2	1,130	1,958
	3	1,237	2,134

(c) Quicksort (number of swaps)

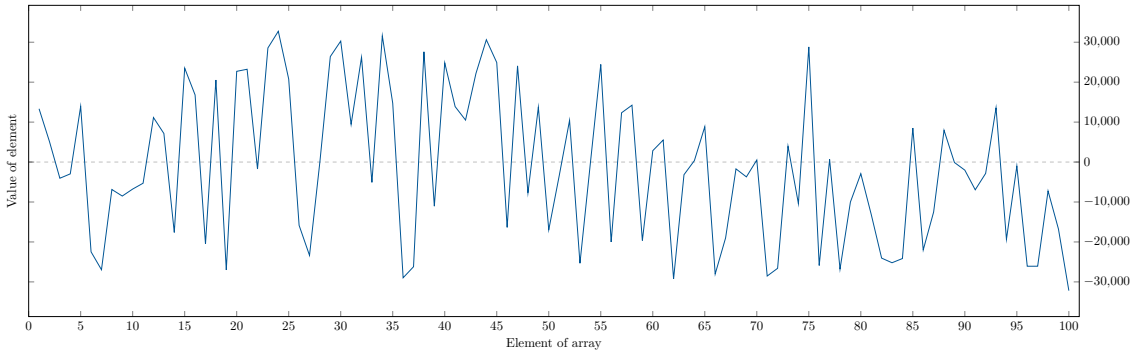
Arguments	Initial sequence	Best sequence
100	145	161
200	336	355

(d) Merge sort (number of comparisons)

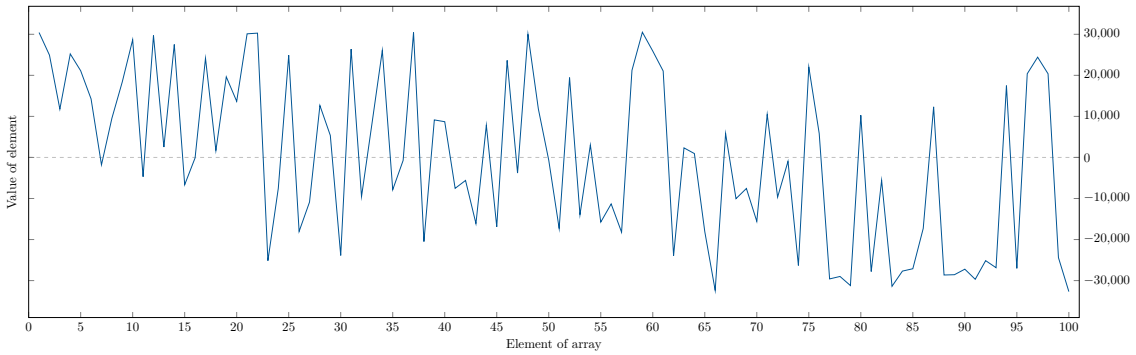
Arguments	Initial sequence	Best sequence
100	540	544
200	1,281	1,287



(a) 1 thread

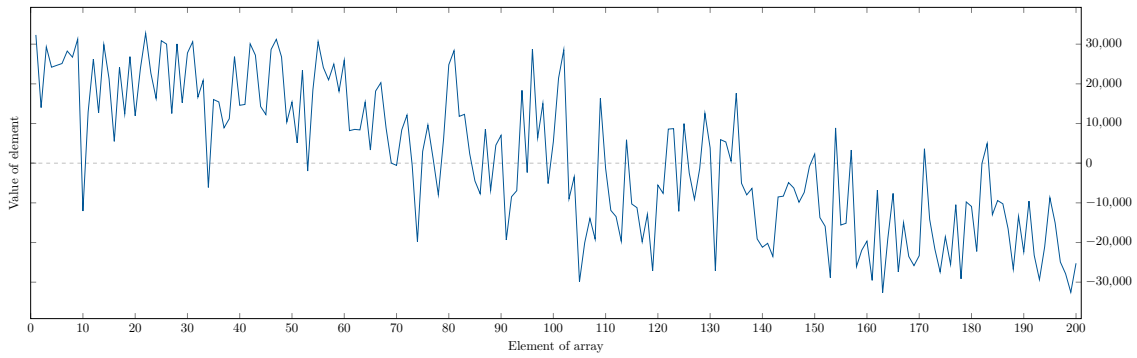


(b) 2 threads

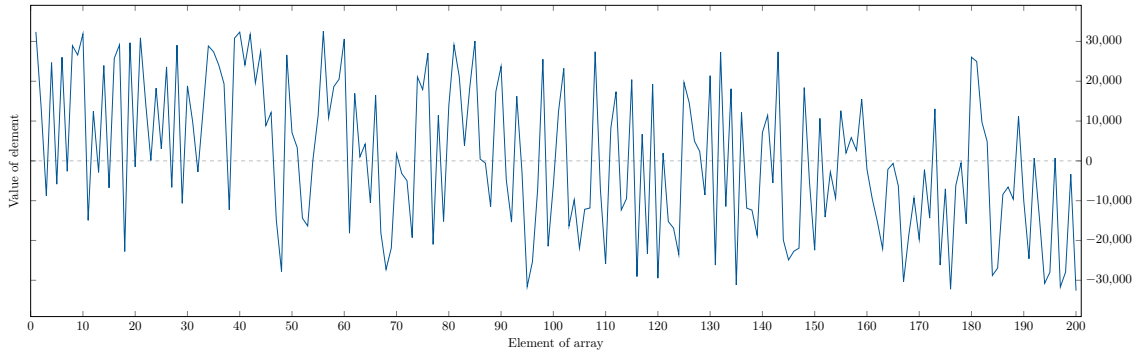


(c) 3 threads

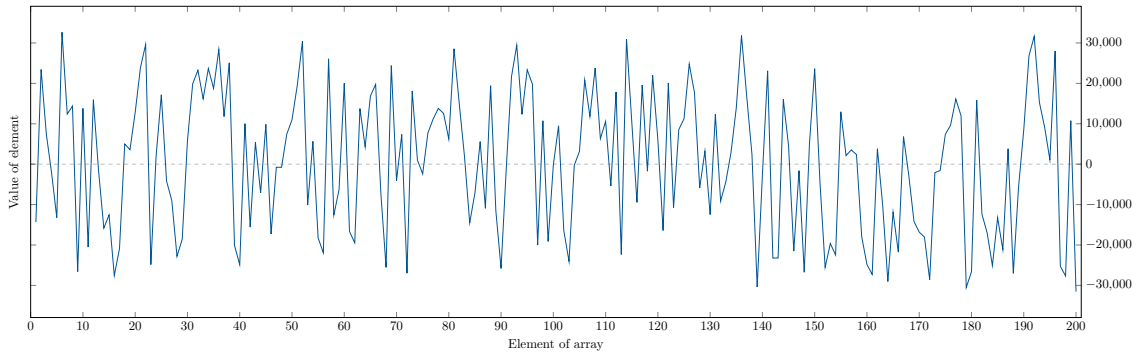
Figure 3.31: Best sequences of 100-element arrays for bubble sort



(a) 1 thread

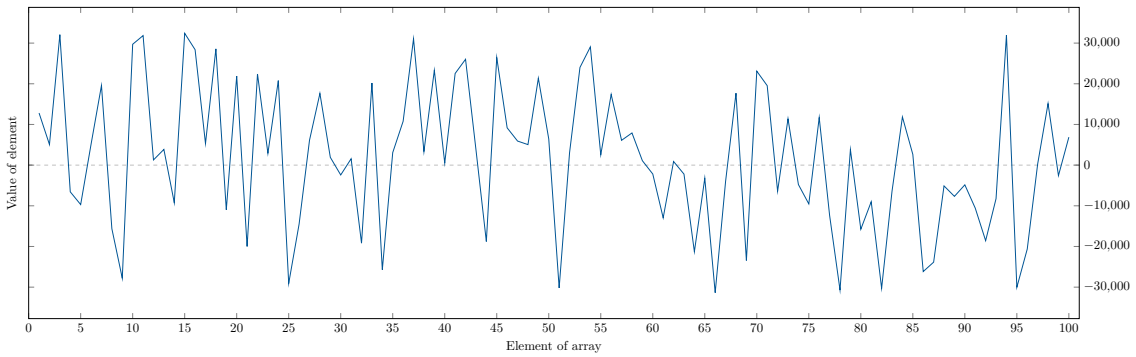


(b) 2 threads

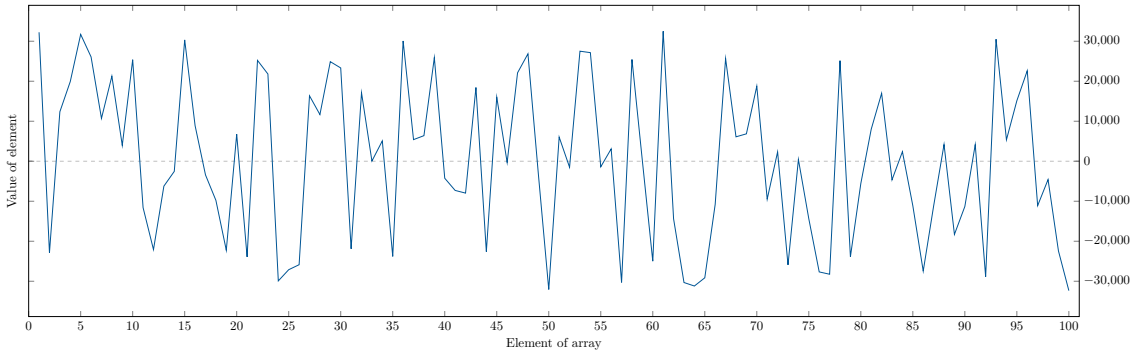


(c) 3 threads

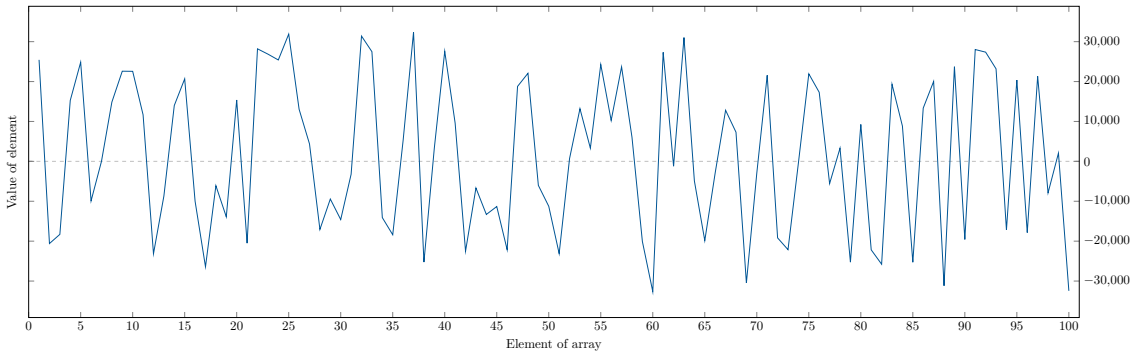
Figure 3.32: Best sequences of 200-element arrays for bubble sort



(a) 1 thread

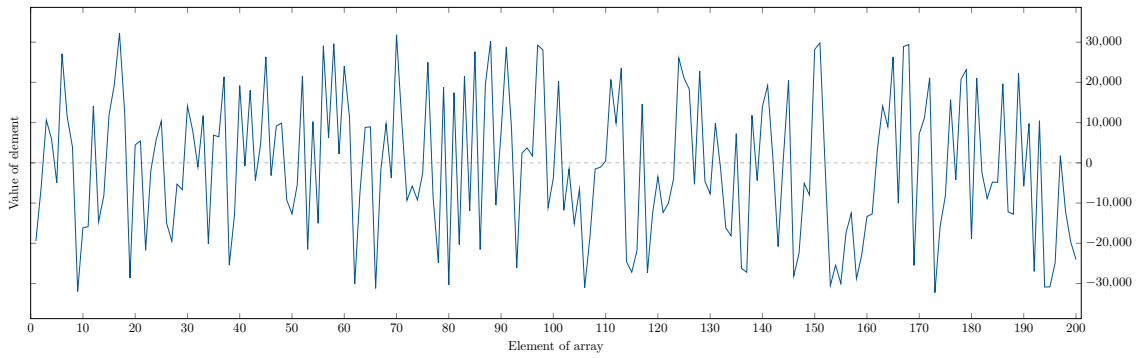


(b) 2 threads

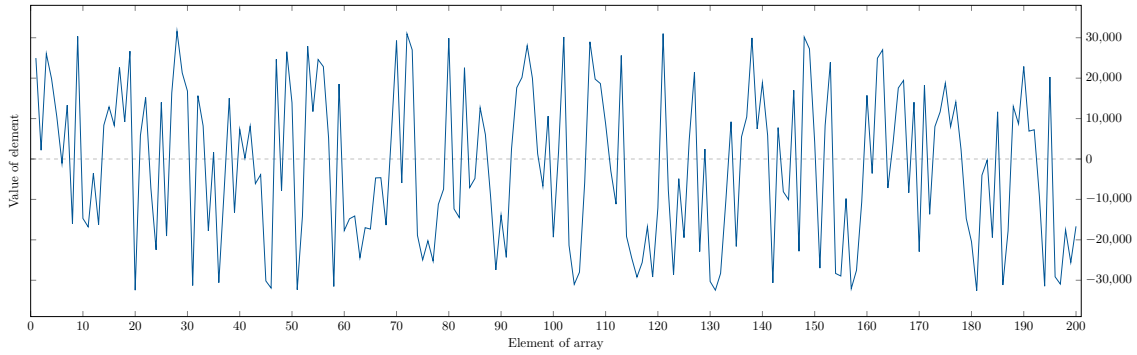


(c) 3 threads

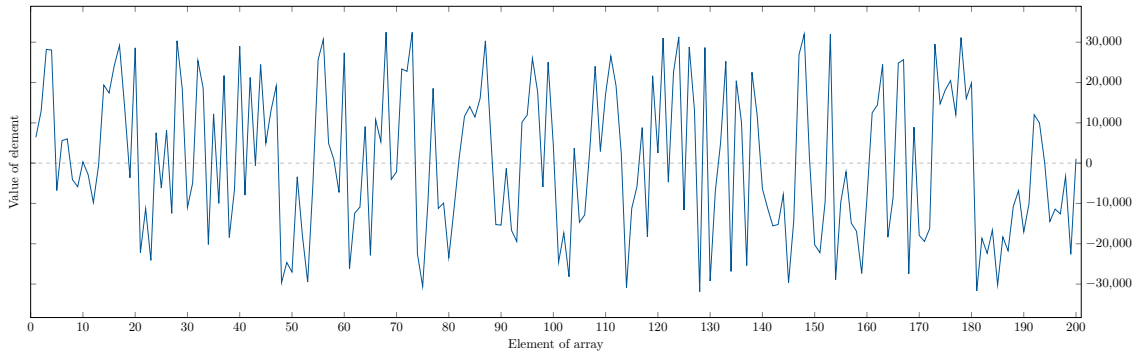
Figure 3.33: Best sequences of 100-element arrays for shell sort



(a) 1 thread

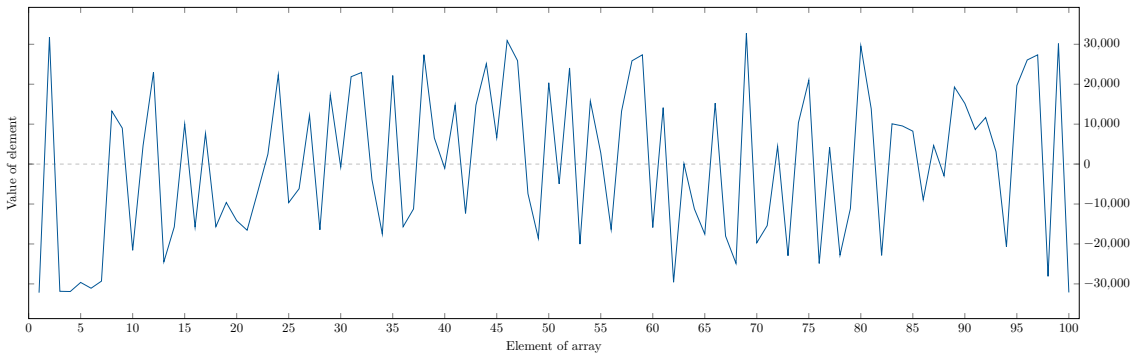


(b) 2 threads

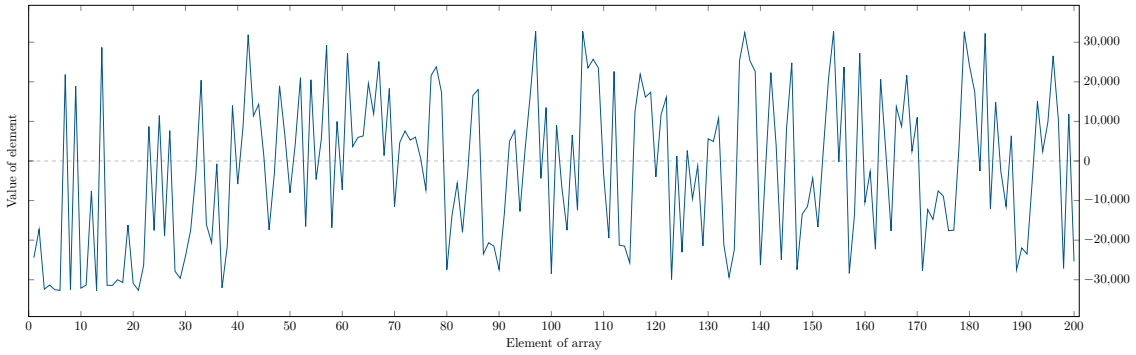


(c) 3 threads

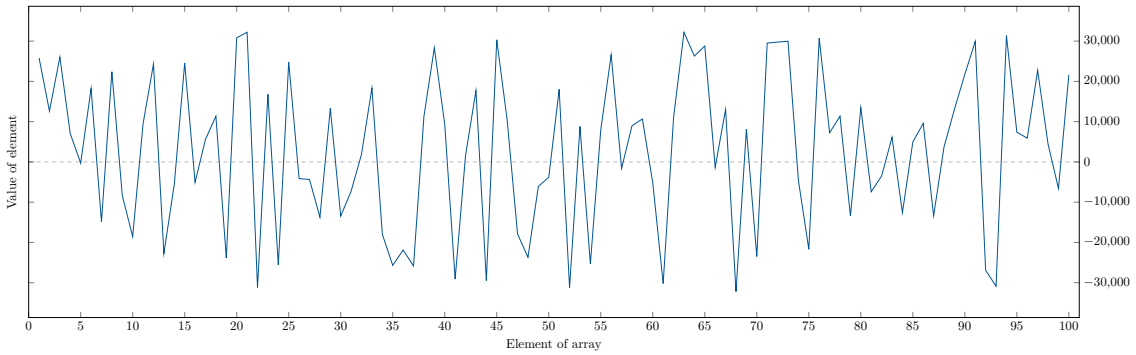
Figure 3.34: Best sequences of 200-element arrays for shell sort



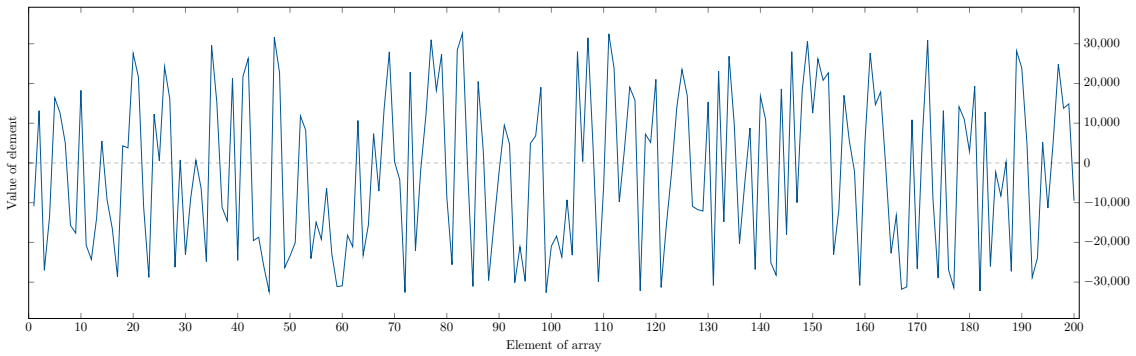
(a) Quicksort—100 inputs



(b) Quicksort—200 inputs



(c) Merge sort—100 inputs



(d) Merge sort—200 inputs

Figure 3.35: Best sequences of 100- and 200-element arrays for quicksort and merge sort

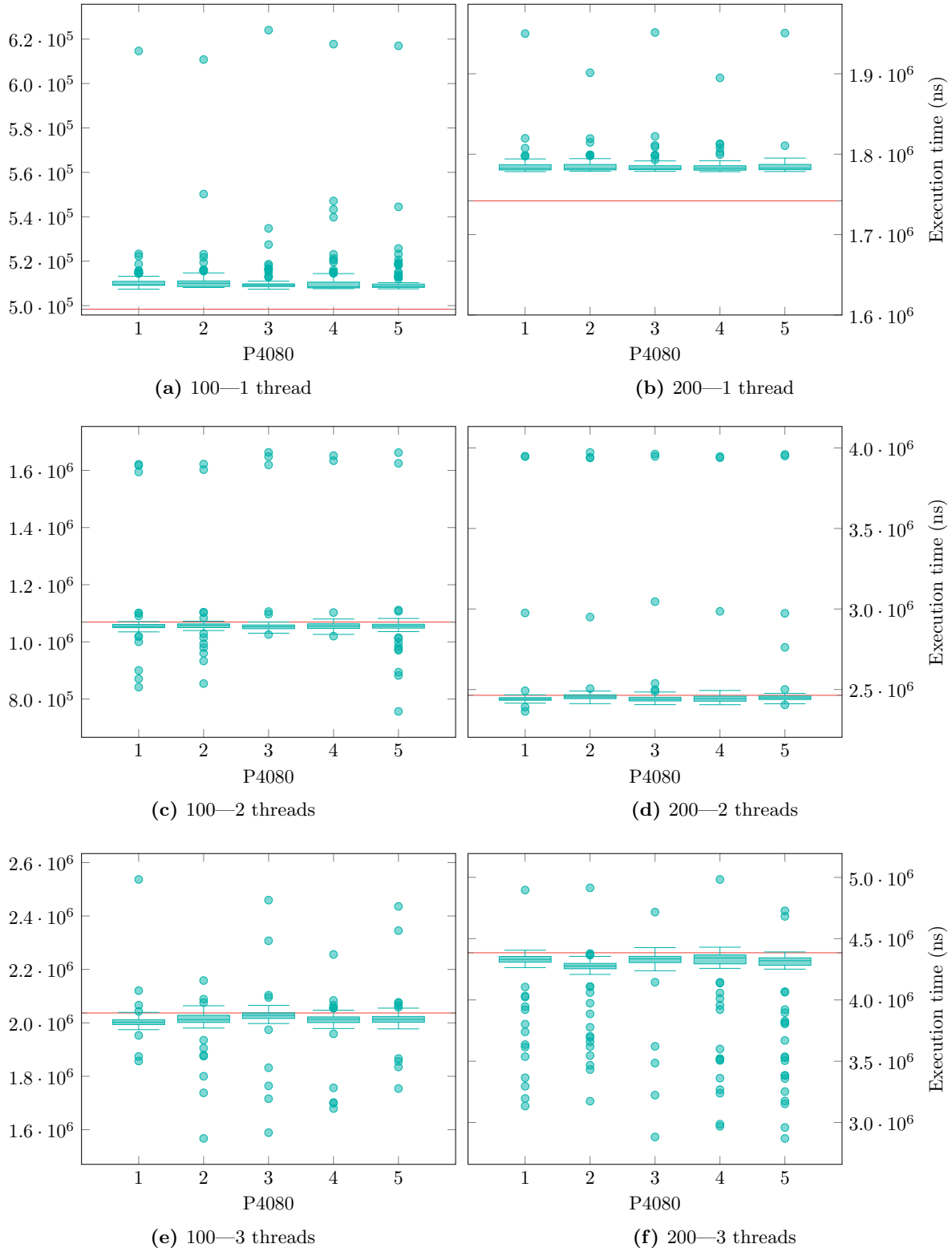
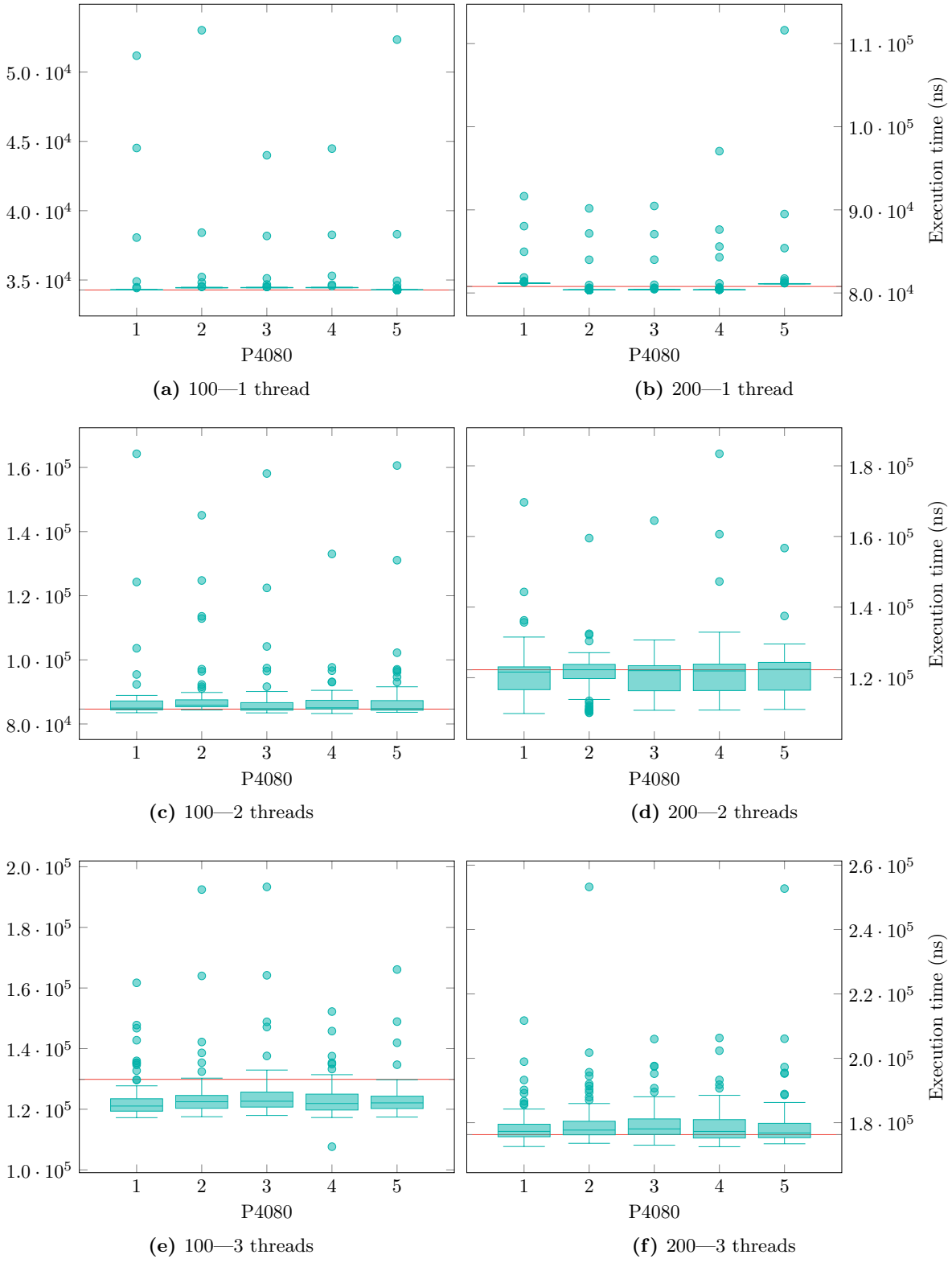


Figure 3.36: Distribution of 100 execution times of bubble sort with 100 and 200 best input arguments obtained from 5 P4080 boards

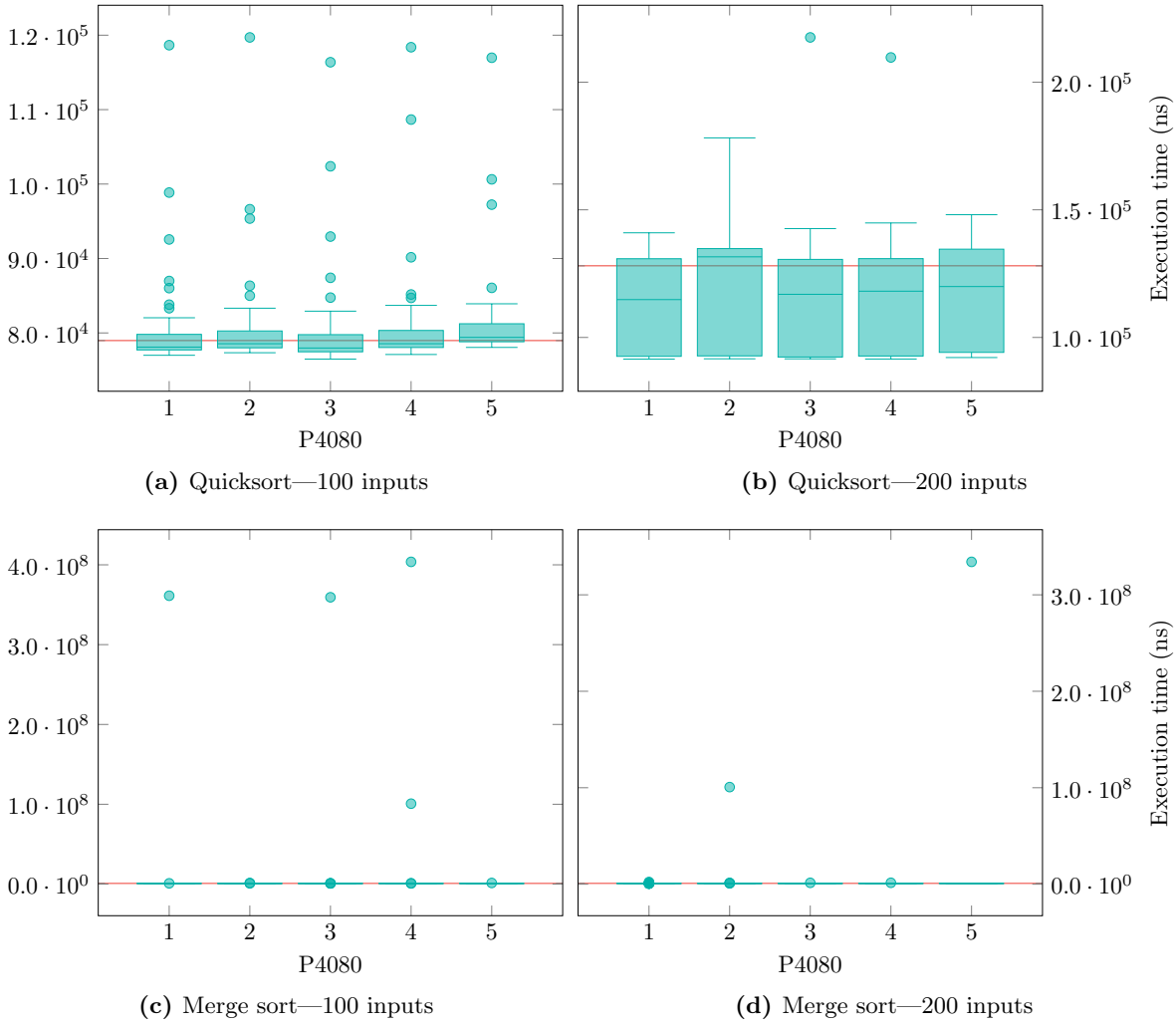


(a) 100—1 thread **(b)** 200—1 thread

(c) 100—2 threads **(d)** 200—2 threads

(e) 100—3 threads **(f)** 200—3 threads

Figure 3.37: Distribution of 100 execution times of shell sort with 100 and 200 best input arguments obtained from 5 P4080 boards



(a) Quicksort—100 inputs

(b) Quicksort—200 inputs

(c) Merge sort—100 inputs

(d) Merge sort—200 inputs

Figure 3.38: Distribution of 100 execution times of quicksort and merge sort with 100 and 200 best input arguments obtained from 5 P4080 boards

Regarding Figures 3.36 to 3.38, overall, there was a small variation or almost no variation of executions times on each P4080 board in almost all the cases, excepting the case of quicksort with 200 input arguments, where the IQR was remarkably broad in all development boards. Besides, among the boards, the IQRs were approximately equivalent in general. As indicated by the (red) lines on the plots, the actual best fitness values were mostly within the ranges of distributions, e.g. some were within the midspreads while others were above. However, as illustrated in Figures 3.36a and 3.36b, the actual best fitness values of the one-threaded bubble sort were significantly outside the distribution ranges of execution times, even though the number of swaps was the same in both the actual experiment and the validation.

Unfortunately, since we only have the best fitness data, we could not further analyse the issue in depth. As previously described in the experimental procedure (Section 3.2.3), to reduce the possible overheads caused by write operations, only the median of 100 replicates, which indeed occurred during the experiment, is written into the output file. Nevertheless, in terms of timing analysis, the actual WCET for a task must be at least as long as any witnessed or measured execution time, as presented in Figure 2.6. The beforementioned experimental best fitness values are just a variation of the execution times of the test inputs under consideration based on the execution environmental conditions at the time the experiment was conducted.

Research Question 3 The third research question is discussed below:

Figures 3.19, 3.20, 3.22 and 3.23 display the distinction of execution times obtained by different numbers of threads. The bar charts show that the execution time was approximately almost double-increased when the number of threads increased, which is not what it is supposed to be. In other words, as the computational problem is broken apart into discrete chunks of work that can be solved simultaneously, a parallel program is expected to increase its performance when more threads are assigned to the cores.

Particularly, suppose we can equally divide the work of our parallel program among the cores. Also, suppose that we run the program with p cores (one thread per each core), and there is no additional work for the cores. Therefore, the parallel program will run p times faster than the serial one. The parallel program is said to have *linear speed-up* when:

$$T_{parallel} = \frac{T_{serial}}{p} \tag{3.6}$$

where $T_{parallel}$ is the parallel run-time and T_{serial} is the serial run-time [114].

In practice, however, we are unlikely to get linear speed-up since the use of multiple threads almost invariably introduces some *overheads*, which are influenced by factors including bus contention, memory contention and cache coherence [124], as previously presented in Section 2.1.4. For example, shared-memory programs (like our sorting benchmarks), which their threads are partitioned and determined either at program creation or as a runtime parameter, will almost always have critical sections [114]. Such critical sections will require the use of some mutual exclusion mechanisms [114], such as mutexes, condition variables and synchronisation barriers. The calls to these functions are overheads that are not present in the serial program, and the parallel program is forced by

such synchronisation operations to serialise execution of the critical section.

It is also likely that the overhead will be increased as the number of threads is increased, i.e. more threads will probably mean more accesses are obliged to a critical section [114]. Hence, it is often the case that:

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead} \tag{3.7}$$

where $T_{overhead}$ is the parallel overhead [114].

Moreover, $T_{overhead}$ often grows more slowly as the *problem size* is increased [114]. Said otherwise, the relative amount of time spent coordinating the work of the threads should be less since there is more work for the threads to do [114]. It might be the case that the small problem input size is the main reason for the odd results of this experiment.

Accordingly, we proved this assumption by running the benchmarks, i.e. bubble sort and shell sort, with different sizes of problem inputs, including 50, 100, 200, 500, 1,000, 2,000 and 5,000, respectively. We also controlled this additional experiment by running the sortings on the same P4080 machine and used the same test inputs generated from `random.org`.

Furthermore, to remove noises as discussed in Section 3.4.1, we used a median of 100 runs to represent the execution time produced by a sorting function. Figures 3.39 and 3.40 illustrate the results of running each benchmark with the different specific number of threads and problem sizes. (Raw data is presented in Appendix C.)

In Figure 3.39, the graphs show that with small problem sizes of 50, 100 and 200, parallel bubble sort with more threads took longer execution time than with fewer threads. But, when the problem size is large enough (e.g. 1,000 inputs), bubble sort with more threads dramatically increased its performance as the execution time is lower than the case of fewer threads. This situation also happened in the case of shell sort as depicted in Figure 3.40; although the larger problem sizes than what we presented here may be required to reveal a significant improvement of parallelism. (The experiment only shows the impact of problem sizes and threads toward a tendency of execution times on parallel programs.)

Therefore, it can be interpreted that shared component resources of a multicore platform may cause interference and result in parallel overhead that is relatively larger than the time spent on the threads themselves if the problem size is too small, and consequently impact the execution time of a task.

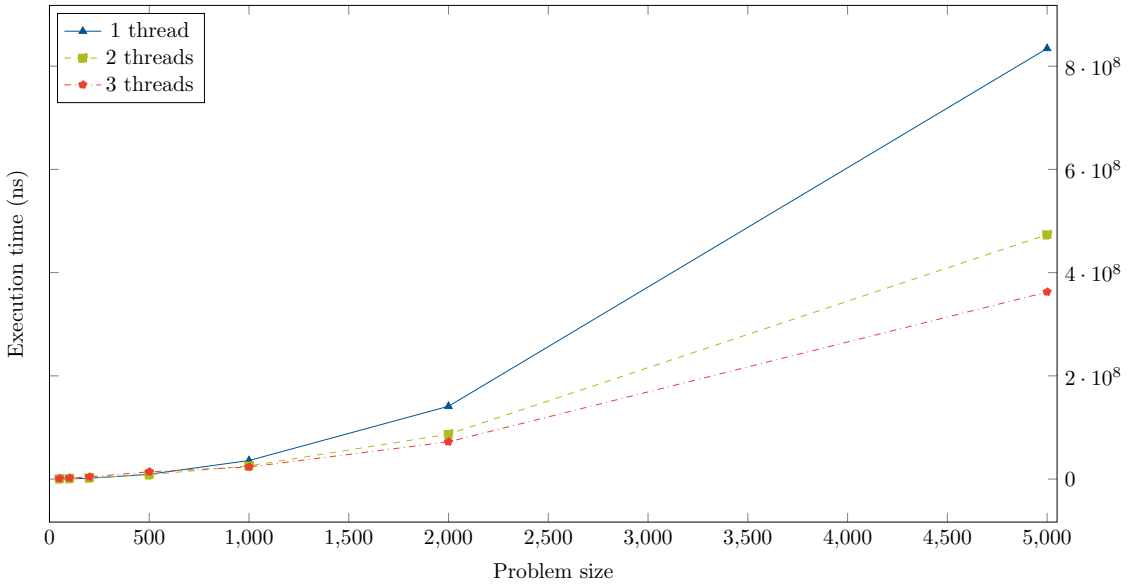


Figure 3.39: Execution times of bubble sort across various problem sizes with the different number of threads

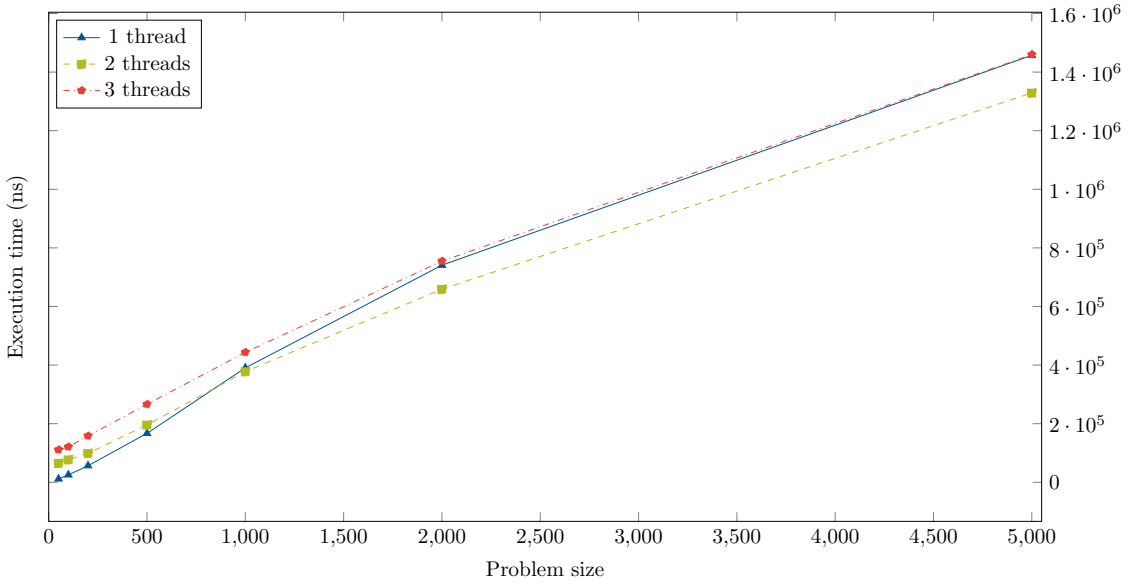


Figure 3.40: Execution times of shell sort across various problem sizes with the different number of threads

3.7 Summary

In summary, this chapter primarily examined the ability of metaheuristic algorithms to seek temporal input arguments that maximise the execution time of a task running on a multicore platform. Based on the computing power of multicore systems, tasks running on them can be classified into single-threaded and multi-threaded tasks. Subsequently, in this chapter, we separated empirical studies into two major experiments: Experiment I—using metaheuristics to find temporal test inputs of a single-threaded function; and Experiment II—using metaheuristics to find temporal test inputs of a multi-threaded function.

Overall, the results from both experiments demonstrate that single-solution based metaheuristics, i.e. HC and SA, and population-based metaheuristic, i.e. GA, were able to effectively find temporal test inputs of numerical functions, including single-threaded polynomial root finder and multi-threaded sortings, in comparison with RS.

Particularly, in Experiment I, where the GSL's polynomial routine was stressed with various number of input arguments (i.e. 5, 7, 9 and 11, respectively) generated by the metaheuristic optimisation algorithms, the results indicate that two single-point metaheuristics (i.e. HC and SA) surpassed the population-based metaheuristic (i.e. GA) for seeking values of the coefficients that maximise the routine's execution time. On the other hand, SHC and RS were the worst and the second-worst performers, respectively. This could be interpreted that SHC was probably too exploitative, while RS was probably too explorative in finding the optimal temporal test inputs on the solution space of such polynomial routine.

In addition, for the cases of larger search spaces of sorting routines with 100 and 200 input arguments in Experiment II, GA was largely found to perform best in several cases and followed by HC and SA, respectively. However, in some cases, either SHC or RS outperformed at least one-third of above-mentioned techniques. It could be pointed out that, with the same number of evaluations for each metaheuristic method to search for optimal temporal test inputs over the larger search space of a sorting routine, there was no explicit outstanding approach, i.e. none approach much better than other.

Accordingly, one may increase the number of evaluations for the search algorithms in the expectation that the longer execution times will be found; still, there is no guarantee that the optimal temporal test inputs could be found as evidenced in Section 3.4.2, since extreme execution times may be concentrated in small partitions of the large input space.

Rather, it would be beneficial if we could reduce the input domain sampled to increase the chances of inputs with extreme times being sampled. Therefore, in Chapter 4, we will present a novel approach named 'dependent input sampling strategies', which can increase the chances of inputs with extreme times being sampled by using a metaheuristic search algorithm to generate parameterised random sampling regimes.

Chapter 4

Indirect Optimisation

4.1 Introduction

4.1.1 Overview

In this chapter, we shift the target of optimisation from test inputs to *strategies* for test input generation. In particular, we target the generation of parameterised random sampling regimes. Search is used to optimise the parameters of a class of such regimes.

The approaches of the previous chapter can be viewed as 'direct' (the test data is the immediate or direct target of the searches) and we will use the term 'indirect' to describe the search for test generation strategies (we search for artefacts which can then be used to generate test data).

4.1.2 Motivation

The outcome of the empirical experimentation in the previous chapter demonstrates that metaheuristic optimisation techniques, i.e. HC, SA and GA, are effective ways of reaching extreme execution times for numeric routines running on an embedded multicore platform. All of them were superior to RS when the search space was small. However, when the problem space was larger, some of them were inferior to RS or even SHC in some cases.

Randomised testing is an important means of testing systems and is an important benchmark for evaluating new approaches. The most common form of randomised testing samples *uniformly* and *independently* from the domains of the inputs. Thus, if the input domains are $D_1, D_2, \ldots, D_n$, then each domain D_i is sampled in turn to produce a test vector $(t_1, t_2, \ldots, t_n)$.

When sampling the domain D_i, all possible elements of that domain have the same chance of being selected. This is what is meant by *uniform* sampling. Furthermore, the sampling of one domain D_i is not affected by any sampling of another domain D_j. This is what is meant by *independent* sampling. This approach is simple, well understood and often easy to implement; though where complex data types are involved, some subtlety may be required.

In ECJ, for example, the `ec.util.MersenneTwisterFast` [85], which is the ECJ's random number generator, uniformly and independently samples a value at random from a given input domain. More specifically, in Experiment I (Section 3.5), each element of the test vector $(a_0, a_1, a_2, a_3, a_4)$ for a quartic equation of $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$ was independently sampled from its own input domain D_{a_0}, D_{a_1}, D_{a_2}, D_{a_3} and D_{a_4}, respectively. Each domain is a uniform distribution $U(-32768, 32767)$.

However, it is far from clear that uniform independent sampling is the most effective approach to discovering test data with extremal properties. Extreme execution times may be concentrated in small partitions of the input space, as illustrated in Figure 4.1. Much of the input space may give unexceptional execution times. It would be beneficial to constrain in some way the sampled inputs to more productive regions in order to increase the chances of inputs with extreme times being sampled. However, we generally cannot identify such regions confidently.

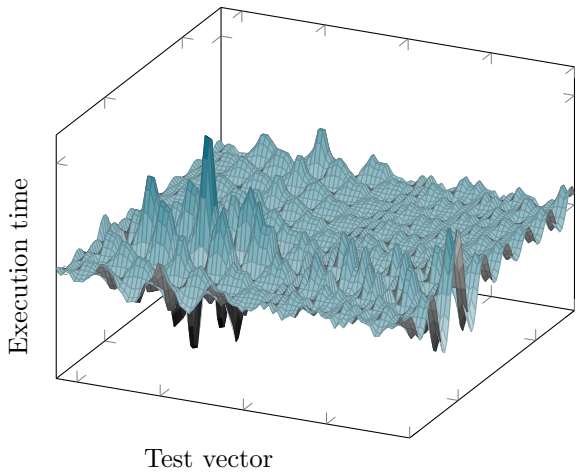


Figure 4.1: Example of search space for test data with extremal properties

Also, suppose that a program has two parameters and its execution incurs extreme times when both parameters are large or when both are small. In a sense, one would want to be able to recognise that a 'good' choice for the second parameter depends on what value was sampled for the first. Thus, we may acknowledge that *dependent* sampling may well offer a more efficient means of gaining appropriate test data in this case.

Accordingly, in this chapter, we investigate an approach that allows the restriction to a subset of the input domain and which also allows the sampling distribution for a parameter to depend on the sampled values of earlier parameters. The idea is to shift the target of optimisation from the test inputs to strategies for test input generation. In particular, we target the generation of parameterised random sampling regimes. A metaheuristic search is used to optimise the parameters of a class of such regimes, which can then be used to generate test data. Our proposed approach can be viewed as an *indirect* approach to test data generation since the metaheuristic operates on the search space of parameterised distributions.

4.1.3 Contributions

The contributions in this chapter are:

- Demonstration of how optimisation can be used to find a dependent input sampling strategy (an approach for generating a test input from a subset of the input domain and where the sampling distribution for a parameter is dependent on the sampled values of antecedent parameters);
- Provision of empirical evidence to show that using the dependent input sampling strategies discovered is a more effective way of sampling test inputs that will give rise to extreme execution times for numerical functions running on an embedded multicore system than a *uniformly independent sampling* approach.

4.1.4 Chapter Outline

The structure of the remainder of this chapter is as follows:

In Section 4.2, a comprehensive description of the proposed approach, i.e. dependent input sampling strategy, is presented.

Section 4.3 gives a preliminary assessment to clarify how particular parameter values are preferred for carrying out the experiments throughout the chapter.

The viability and performance of the proposed approach with a basic interval for extremal timing performance of a (single-threaded) numerical function are then demonstrated in Experiment III (Section 4.4).

Experiment IV (Section 4.5) explores the performance of the approach with a fixed delta interval, which is a fixed parameter value to specify the upper bound for intervals of the input subdomains, for extremal timing performance of the numerical function.

Experiment V (Section 4.6) examines the performance of the approach with a randomised delta interval, which is a randomised parameter value sampled from a given range to specify the upper bound for intervals of the input subdomains, for extremal timing performance of the numerical function.

The results obtained by those three main experiments are altogether discussed in detail in Section 4.7.

Finally, in Section 4.8, the results of the empirical research in Experiments III, IV and V are summarised.

4.2 Dependent Input Sampling Strategies

4.2.1 Definition

A priori, it is generally unclear what test inputs will give rise to extremal behaviour, and it is far from clear that sampling uniformly and at random is a good way of discovering test data with extremal properties. Our investigation of dependent approaches is based on the assumption that sampling uniformly across a full domain is unlikely to be particularly efficient to reveal optimal test inputs in temporal test vectors. Two clear means of improving efficiency suggest themselves: allowing subdomains to be sampled rather than the full domain; and allowing such subdomains to be sampled non-uniformly.

Also, in many cases, there *may* be an *implicit* assumption that the sampled domain is in some sense contiguous, e.g. integer values sampled from a single full domain of all 2^{32} possible `int` values by Java's `nextInt()` method. However, this again does not seem essential. It would be perfectly possible (and implementation-wise quite easy to effect) to have a sampling domain that was the union of two or more disjoint domains.

Specifically, we will not insist on whole domain sampling. The sampling subdomain for any parameter will be a union of a number of domain sub-ranges, e.g. if the full input domain is $[-100, 100]$, we might have $[-32, 17] \cup [54, 98]$ as the domain sub-ranges actually sampled. We would not know in advance which constituent intervals work best but will seek to discover this as part of the search. We will not insist on subdomains being disjoint. It is feasible that overlapping domain sub-ranges for a parameter may give better results. We should, therefore, allow it and allow the search process to discover whether disjoint or overlapping domain sub-ranges are best. Particularly, in order to sample from the subdomain of a parameter, we first select one of its intervals (i.e. the domain sub-ranges) in accordance with some probability distribution and then sample uniformly from that selected interval. Hence, the probability of a value being selected depends on not only the probabilities of intervals containing that value being selected but also the sizes of those intervals. A value that is in the intersection of two intervals has a greater likelihood of selection than a value that is in a single interval alone. For instance, it follows that $[0, 2] \cup [0, 2]$ gives rise to different sampling distribution than $[0, 1] \cup [0, 2]$, even though the overall sampled domain is $[0, 2]$. In the first, the three values (i.e. 0, 1 and 2) are sampled with equal probability, whereas in the second the value 1 has the greatest chance of being selected (assuming interval selection probabilities are non-zero).

Moreover, we will also allow the sampling distribution of an input subdomain to depend on the values of subdomains sampled earlier. For example, in regard to the example of input domains for the test vector $t_1, t_2, \ldots, t_n$ as previously described in Section 4.1.2, the sampling distribution of D_3 will depend on the sampled values (i.e. the selected domain sub-ranges) of D_1 and D_2. More precisely, let d_{i_j} be a domain sub-range of D_i, so that a finite union of sub-ranges of D_i is:

$$\bigcup_{j=1}^{n} D_{i_j} \tag{4.1}$$

The sampling of a parameter t_i will depend on the specific intervals d_{i_j} sampled for previous parameters, rather than the exact values sampled from those chosen domain sub-ranges. The overall sampling regimes can be seen as a tree.

Figure 4.2 shows an example of a tree representation of a sampling distribution for a three-parameter (A, B, C) problem. For simplicity, we assume here that the sampling domain for each parameter is the union of two sub-ranges as indicated above. In particular, we assume that the first parameter sampled A comprises two intervals $[l_{A_0}, u_{A_0}]$ and $[l_{A_1}, u_{A_1}]$. The weights w_{A_0} and w_{A_1} represent the relative chances of each interval of A being selected. We will normalise these weights to get the specific probabilities of choosing each of these intervals by using a histogram-based selection method—the proposed algorithm listed in Algorithm 9.

Besides, at each level of the tree, we can see that similar sampling regimes are available for the second parameter sampled B, and so on, but that separate regimes are in place depending on whether the left or right branch was chosen for A. Accordingly, we named the proposed approach 'dependent input sampling strategies'.

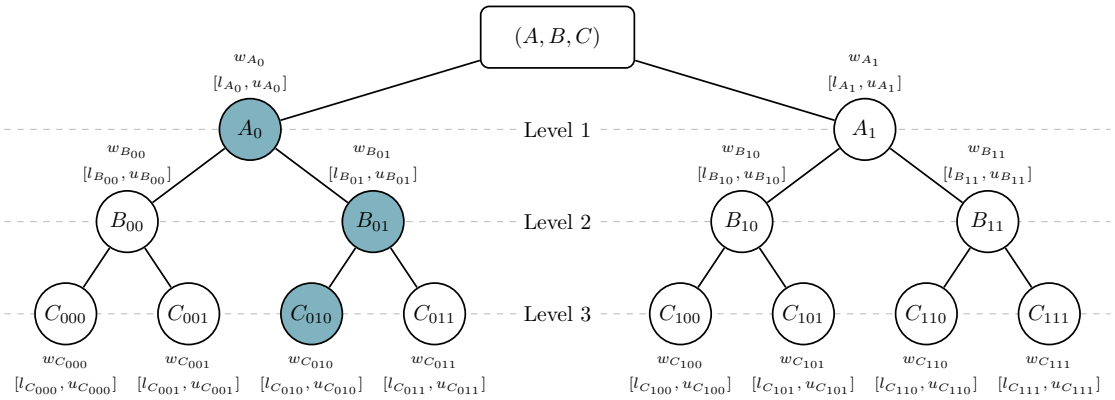


Figure 4.2: Example of a tree structure for a sampling distribution

Algorithm 9: Histogram-based selection method

$n \leftarrow$ a given total number of intervals of each parameter
$W = \{w_0, w_1, w_2, \ldots, w_{n-1}\}$, a set of weights
for all w_i such that $w_i = 0$ **do**
  set $w_i = 1$ ▷ This imposes equal probabilities in this degenerate case and avoids division by zero under normalisation below
end for
$total = \sum_{i=0}^{n-1} w_i$, a summation of weights
$normalised \leftarrow \varnothing$, an initial set of normalised weights
$cumulative \leftarrow \varnothing$, an initial set of cumulative weights
for $i = 0$ **to** $n - 1$ **do**
  $normalised_i = \frac{w_i}{total}$
  if $i = 0$ **then**
    $cumulative_i = normalised_i$
  else
    $cumulative_i = normalised_i + cumulative_{i-1}$
  end if
end for
$index \leftarrow 0$, an initial index
$r \leftarrow U(0, 1)$, a random number sampled from the uniform distribution
while $r > cumulative_{index}$ **do**
  $index = index + 1$
end while
return *index*

According to Figure 4.2, suppose that domain sub-ranges A_0, B_{01} and C_{010} are selected, respectively, as highlighted in green. Therefore, the test vector of the three-parameter (A, B, C) problem will be sampled from such selected intervals, i.e. $[l_{A_0}, u_{A_0}]$, $[l_{B_{01}}, u_{B_{01}}]$ and $[l_{C_{010}}, u_{C_{010}}]$.

Regarding the tree representation described above, given that I is a set of input parameters (or a test vector) for a given problem, $|I|$ is the cardinality of I (i.e. the number of tree levels) and c is the number of domain sub-ranges per each parameter.

Thus, the total number of all possible dependent domain sub-range sets for the problem or sampling regimes (i.e. the number of tree paths), denoted P, is:

$$P = c^{|I|} \tag{4.2}$$

and the total number of all possible domain sub-ranges that can be sampled (i.e. the number of tree nodes), denoted N, is:

$$N = \sum_{i=1}^{|I|} c^i \tag{4.3}$$

For instance, in Figure 4.2, $I = \{A, B, C\}$, $|I| = 3$ and $c = 2$; so that $P = 8$ and $N = 14$.

4.2.2 Implementation

Rather than the use of notable optimisation techniques to directly generate test inputs in the same way as in Chapter 3, as well as in all the previous work in search-based temporal testing, a candidate solution of such metaheuristic is used to construct a tree representation of a sampling distribution for our novel approach. The test inputs are then sampled from the dependent domain sub-ranges of any regime path of the constructed tree.

To this end, we utilised the ECJ toolkit [81] to generate a solution genome, where its elements are decoded to the associated parameters of the tree's nodes, including intervals and weights. The metaheuristic search will operate on evolving parameterised random sampling regimes for our indirect approach. Beside the metaheuristics' parameters for such parameterised generation, to further manipulate the mechanism of the indirect approach for sampling test data, three additional parameters are necessitated, i.e. the number of sub-ranges, the type of interval and the number of sampling trials.

Number of Sub-Ranges A tree representation of a sampling distribution can be constructed in various hierarchical forms depending on the number of sub-ranges designated to each problem parameter. The number of domain sub-ranges defines the *diversity* of the approach over the search space of test inputs (exploration). Particularly, the higher number of sub-ranges per each parameter induces the approach to explore the temporal test data on more different places over the search space. In other words, there are more paths of the tree to be selected for generating a test vector.

Type of Interval Regarding the tree structure described earlier, each node (or a sub-range) comprises three elements, i.e. a finite lower and upper bounds of interval and selection weight. Such sub-range forces the dependent approach to *intensify* over a small partition of the input space (exploitation). The approach classifies the interval into three: basic, fixed delta-based and randomised delta-based types, which will be presented later in this chapter (Sections 4.4 to 4.6).

Number of Sampling Trials After the tree is formed, i.e. all sub-ranges' elements are decoded from the solution genome, in order to empower our approach to explore and exploit the search space as described above, the test vectors are generated based on such established sampling distribution for the number of trials. On each trial, the test vector can be generated from any path of the tree (a selected sampling regime). It is not necessary to stick the test data generation with any specific tree path for all trials. Then, the highest fitness value of the test vector produced from among the number of trials represents the fitness value of such constructed tree (or the solution genome). After that, the metaheuristic algorithm repeats the process by evaluating the fitness value of such solution genome and then generating newly parameterised random sampling regimes. The new tree is, therefore, reconstructed by such newly evolved solution genome. The process continues until the terminational conditions are satisfied.

4.3 Preliminary Analysis

As earlier described in Section 4.2.2, there are three supplementary parameters for dependent input sampling strategies, including the number of sub-ranges, the interval type and the number of sampling trials.

For simplicity, we first permanently set the number of sampling trials to ten for all experiments in this chapter because such value allows the approach to be explorative enough at each fitness evaluation of the utilised metaheuristic while consumes an acceptable computational time. (Further insight into a benefit-cost ratio of the number of sampling trials is the subject of future work.) We hence only performed a preliminary assessment to choose the suitable number of sub-ranges to be used for the rest of the experiments. Finally, the three main experiments in this chapter are carried out based on different interval types.

4.3.1 Number of Sub-Ranges

The number of sub-ranges for each problem parameter can be separately assigned with different values. For instance, regarding the three-parameter (A, B, C) problem as previously described in Section 4.2.1, the number of sub-ranges c_A, c_B and c_C could be assigned to 2, 3, and 4, respectively. However, for simplicity, we designated the same number of sub-ranges to all problem parameters.

The preliminary experiments were conducted based on four different numbers of sub-ranges, i.e. 2, 3, 5 and 10, respectively. Accordingly, we followed the procedure that was previously done in the preliminary analysis of Section 3.4. In this chapter, however, GA was used for generating sampling regime trees instead. Furthermore, since there are three different types of intervals, we simply used the basic one for the preliminary study. Thus, three elements of each sub-range, i.e. a lower bound, an upper bound and a selection weight, were evolved by GA. (Further details on the basic interval type can be found in the next section.) Figure 4.3 depicts the results obtained from running GA on the quartic equation with different numbers of sub-ranges.

As shown in Figure 4.3, two sub-ranges was superior to every other number of sub-ranges, while three sub-ranges was the worst. Based on the given number of sampling trials (i.e. ten trials per constructed tree), it could be assumed that a small number of sub-ranges per problem parameter is sufficient to explore the search space. However, increasing the number of sampling trials may authorise a greater number of sub-ranges to expand its ability on traversing and finding the better sampling regimes from the tree. For instance, there are 2^5 possible sampling regimes over the tree structure for two sub-ranges, while 10^5 sampling regimes for the case of ten sub-ranges. For proof of concept, we picked the values of two and three as a 'good enough' sub-ranges for the rest experiments of this chapter. We targeted to further study the effect of the different number of sub-ranges to the indirect approach as well.

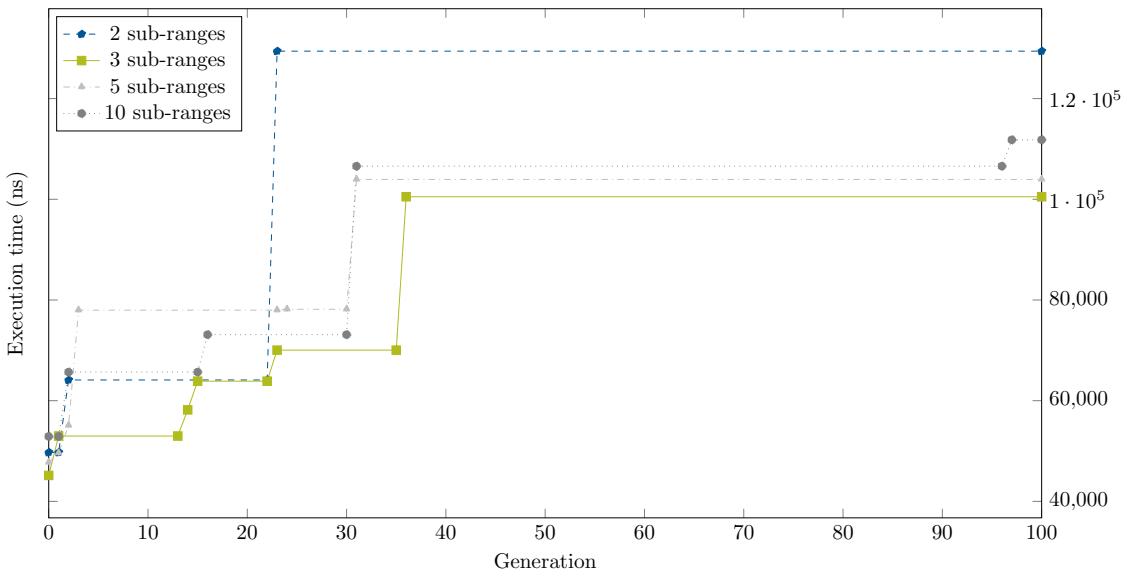


Figure 4.3: Results obtained by GA to the quartic equation with different numbers of sub-ranges

4.4 Experiment III—Basic Interval

4.4.1 Objective

As previously described in the motivation above (Section 4.1.2), reducing the input domain of each test input would be advantageous to increase the opportunities of inputs with extreme times being sampled. In this experiment, we apply a dependent input sampling strategy, which is used in combination with metaheuristic algorithms, to search for test inputs that might cause a numeric function running on the COMX-P4080 development board to violate performance timing requirement. The interval type used in this experiment is a basic one, so both it lower and upper bounds are decoded from the solution genome of the metaheuristic search. Furthermore, the effectiveness of the proposed strategy is able to reflect the ability of a metaheuristic algorithm to seek subdomains (i.e. the intervals) that are likely to encompass inputs with extreme times. Therefore, we apply HC, SA and GA to find such extreme subdomains. In addition, for proof of concept, we evaluate our proposed approach with the GSL's polynomial root-finding routine. Hence, the problem statement in this experiment is similar to Experiment I in Chapter 3, which is:

Problem Statement: *For a polynomial equation of the form* $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$*, we seek values of the coefficients (*a_0 *to* a_n*) that maximise the execution time of the polynomial solver.*

The research questions addressed in this experiment are:

Research Question 1: *Is the dependent input sampling strategy with a basic interval effective in seeking the values of the coefficients that maximise the execution time of the polynomial solver?*

Research Question 2: *Among the metaheuristics, which technique is the best for supporting the dependent input sampling strategy with a basic interval to seek test inputs that maximise the execution time of the polynomial solver?*

Research Question 3: *Does the number of interval choices affect the effectiveness of the dependent input sampling strategy with a basic interval?*

4.4.2 Preparation

Since the metaheuristic techniques are utilised to assign relevant values, i.e. an interval and weight, to each sub-range choice (or a node of the tree structure), those ECJ's features earlier used in Chapter 3 were also employed in this experiment. We excluded SHC because of its neighbours' definition (stated in Section 3.2.1), together with its acceptance criterion (listed in Algorithm 3), which may restrict the variation of values given to the sub-range.

The parameter settings for these metaheuristic algorithms are also the same as in the preceding chapter (listed in Table 3.2). However, for a basic interval type of indirect approach, the `genome-size` given to the ECJ's metaheuristic algorithms was slightly different from those in Experiment I (Section 3.5). Specifically, the size of the genome is triple the size of N (recall that N is the total number of all possible domain sub-ranges), as three elements, i.e. a lower and upper bounds of an interval, and a selection weight, are required to be decoded to each sub-range of a sampling distribution.

Figure 4.4 illustrates the genome size of 42 for the three-parameter (A, B, C) problem with a union of two sub-ranges per each parameter from Figure 4.2. A value stored in each gene of the genome is decoded to each node of the tree data structure by using a combination of two tree traversals, i.e. level-order and pre-order, respectively. Other tree representations are not prohibited. Indeed, the approach here is one of many. An optimal representation is the subject of future research.

In this experiment, the polynomial routine was executed with two different numbers of input arguments, i.e. 5 and 7, respectively, along with two variant numbers of sub-ranges per each input argument. Accordingly, the genome sizes given to the ECJ are summarised in Table 4.1.

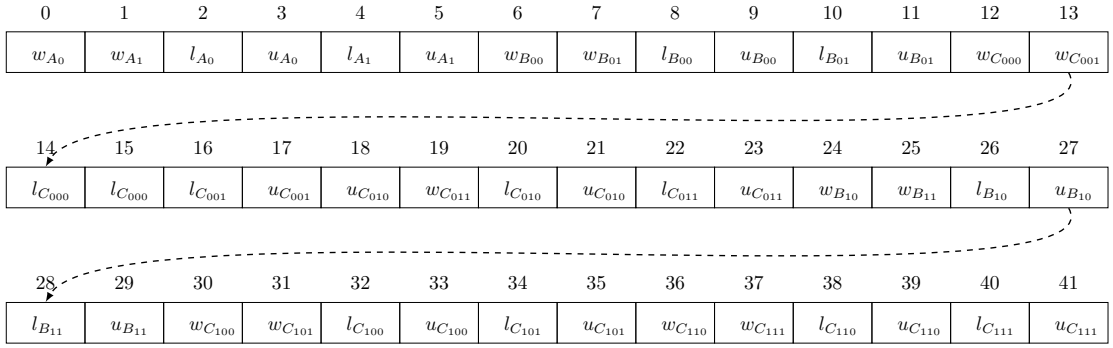


Figure 4.4: Example of a solution genome of the dependent input sampling approach with a basic interval

Table 4.1: Genome sizes for the empirical experiments of the dependent input sampling approach with a basic interval

Arguments	Sub-ranges	Genome size
5	2	186
	3	1,089
7	2	762
	3	9,837

4.4.3 Method

In the same way as in Experiment I (Section 3.5), to perform four empirical experiments listed in Table 4.1, each metaheuristic technique was executed with the polynomial routine for ten trials. An initial seed obtained from `random.org` for each metaheuristic technique is listed in Table 4.2.

In the proposed indirect approach, in particular, an individual or a candidate solution genome generated by a metaheuristic algorithm is used to constructing a tree as described earlier in Section 4.2. For proof of concept and simplicity as stated in the preliminary analysis, we took ten sampling trials per each decoded tree. In other words, any tree representation will be used to generate ten test data samples. The sampled test vector that gives the highest execution time will be stored and its execution time will be used to represent the fitness value of the individual tree. The metaheuristic algorithm will then continue its search process to find more suitable trees, i.e. sampling distributions. Consequently, the computation time of each experimental trial is approximately ten times longer than the one in Experiment I; it takes around 30 hours per experimental trial.

Table 4.2: Initial seeds of metaheuristic algorithms for different input arguments of the polynomial root-finder (Experiment III)

(a) 2 sub-ranges

	Arguments	
Algorithm	**5**	**7**
HC	−23,990	11,717
SA	32,687	−8,936
GA	18,254	−5,390

(b) 3 sub-ranges

	Arguments	
Algorithm	**5**	**7**
HC	21,009	−27,595
SA	−6,845	13,575
GA	13,854	17,629

4.4.4 Results

The results from Experiment I in Chapter 3 are used as a baseline for comparisons with the results of this experiment and the subsequent empirical studies in this chapter. For ease of reference, we refer to those previous results of Experiment I by prefixing 'direct' to the metaheuristics' names, whereas 'indirect' is prefixed to the names of metaheuristics for the indirect approach results. Also, a value in the bracket indicates the number of sub-ranges per each test input of a test vector.

Figures 4.5 and 4.6 depict the differences between the initial and the final (best) fitness values gained from ten trials of each experiment on stressing the polynomial routine with two different numbers of input arguments, i.e. 5 and 7, respectively.

Distribution of the best fitness values gained from ten trials of each experiment to stress the polynomial solver is illustrated in Figures 4.7 and 4.8.

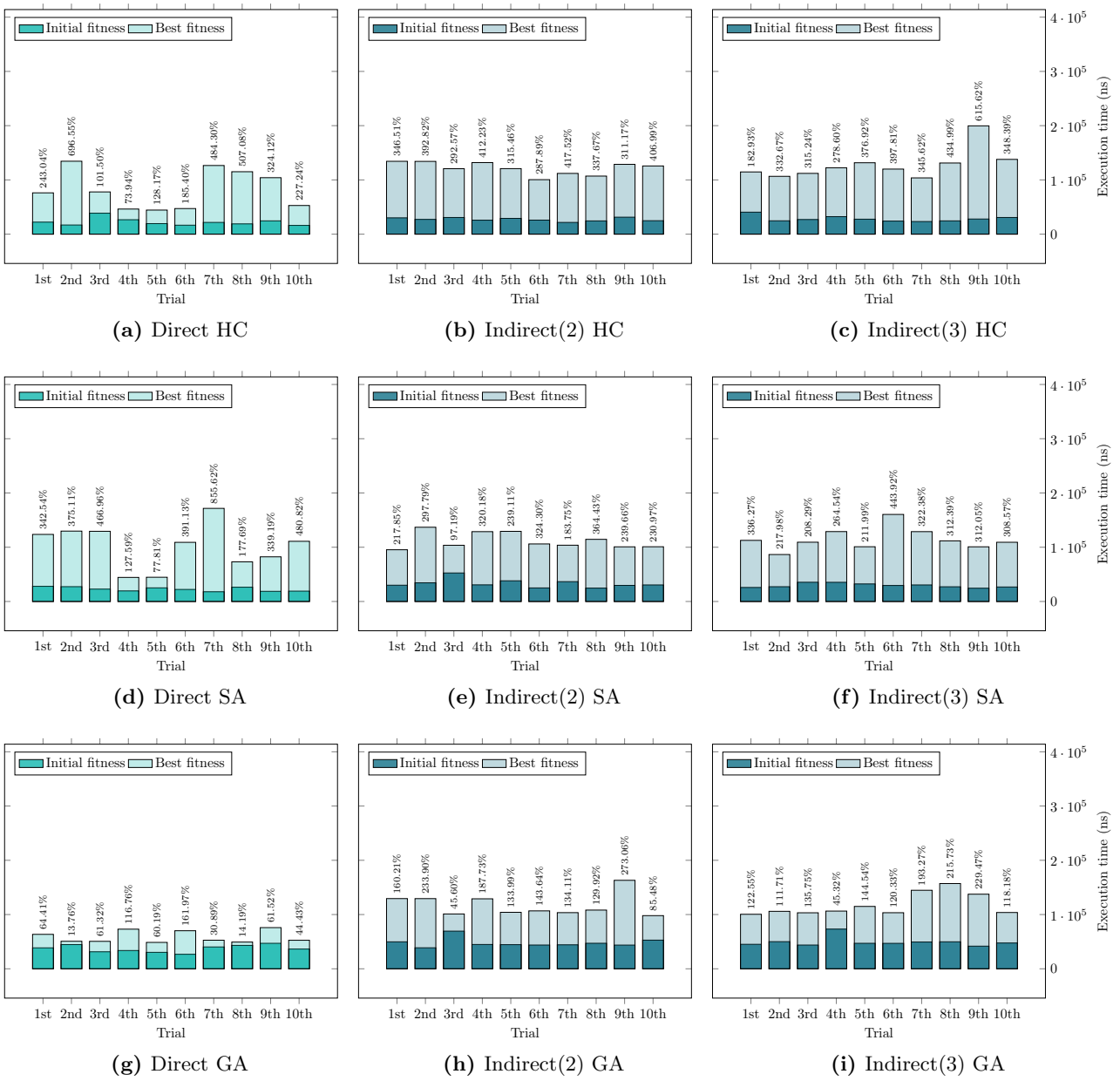
Finally, the longest execution time among ten trials of each experiment is summarised in Table 4.3.

4.4.5 Discussion and Conclusions

Research Question 1 In order to answer the first research question of this experiment, we assessed the effectiveness of the dependent input sampling strategy with a basic interval on the ability to seek values of the coefficients that maximise the execution time of the polynomial solver by comparing its results with the ones from Experiment I.

Particularly, the fitness values in Experiment I were gained from the execution of the polynomial solver and the test vector, where each test input was uniformly and independently sampled from the input domain $[-32768, 32767]$. The fitness values in this experiment were, on the other hands, obtained by executing the polynomial solver with the test vector, where each test input was dependently sampled from selected sampling regime.

According to Figures 4.7 and 4.8, overall, both of the dependent input sampling strategies, i.e. two and three sub-ranges, performed more stable over ten experimental trials than the direct ones. However, in several cases, the direct metaheuristic technique delivered more extreme execution times than the dependent input sampling strategy with a basic interval, such as SA in quartic equation case, and HC and SA in sextic equation case, as shown in Table 4.3.

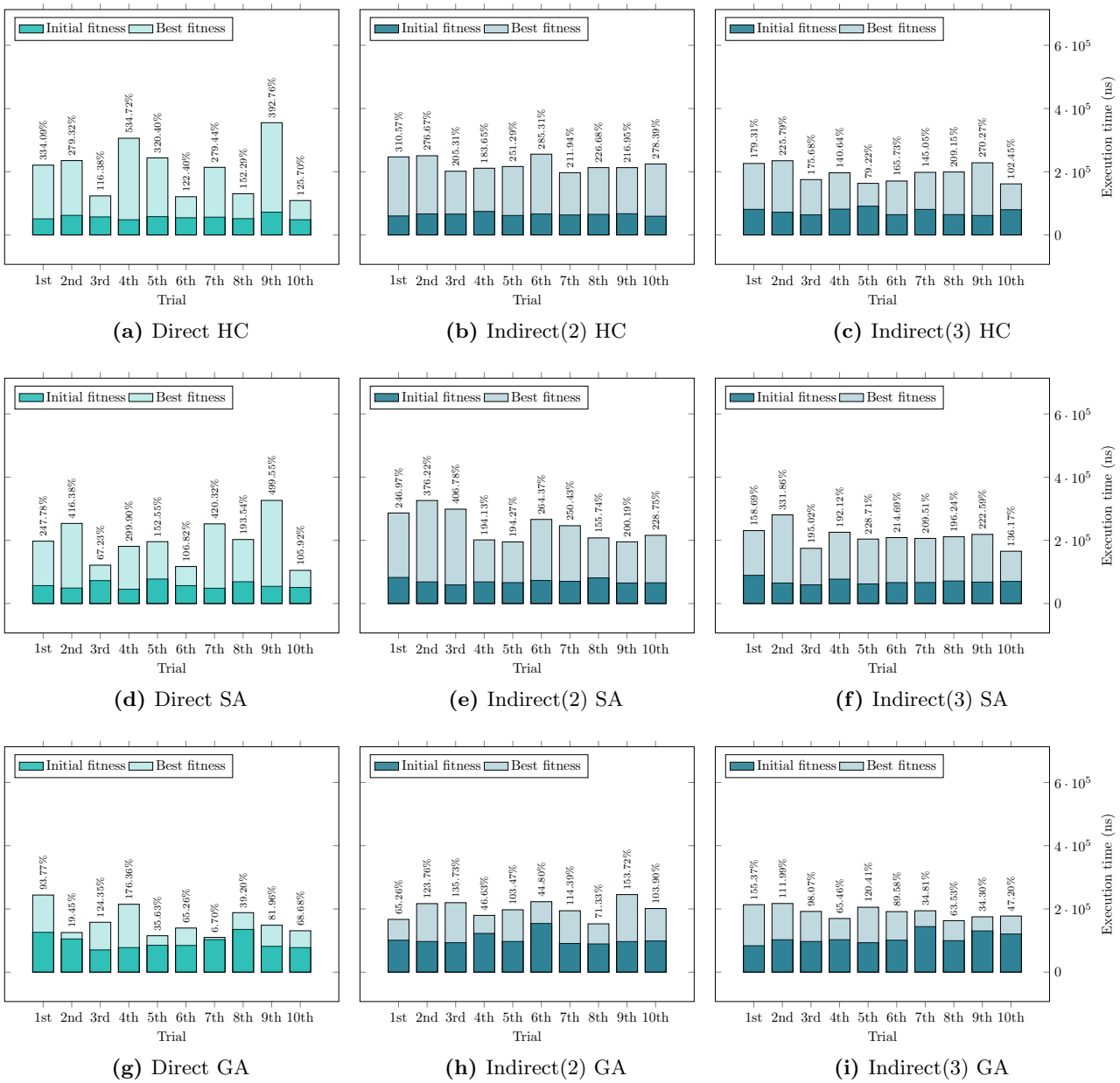


(a) Direct HC **(b)** Indirect(2) HC **(c)** Indirect(3) HC

(d) Direct SA **(e)** Indirect(2) SA **(f)** Indirect(3) SA

(g) Direct GA **(h)** Indirect(2) GA **(i)** Indirect(3) GA

Figure 4.5: Results of 10 trials obtained by the basic indirect approaches to the quartic equation in comparison with the direct approaches

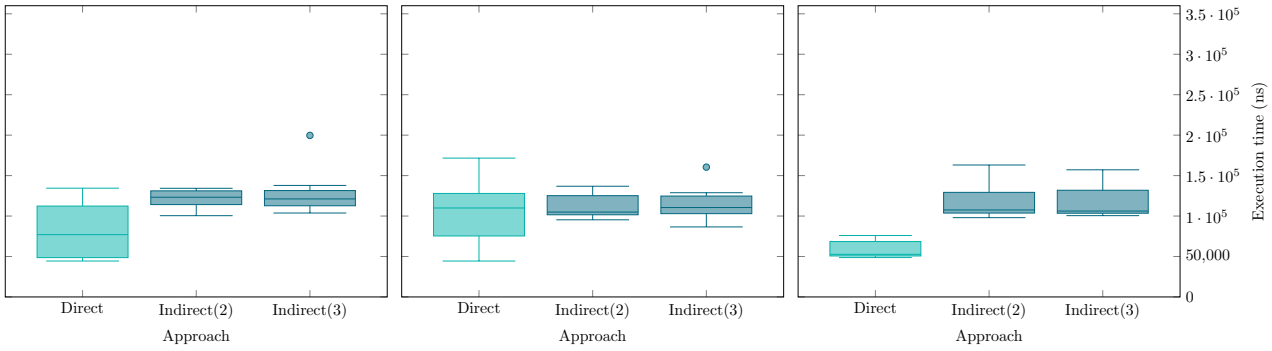


(a) Direct HC **(b)** Indirect(2) HC **(c)** Indirect(3) HC

(d) Direct SA **(e)** Indirect(2) SA **(f)** Indirect(3) SA

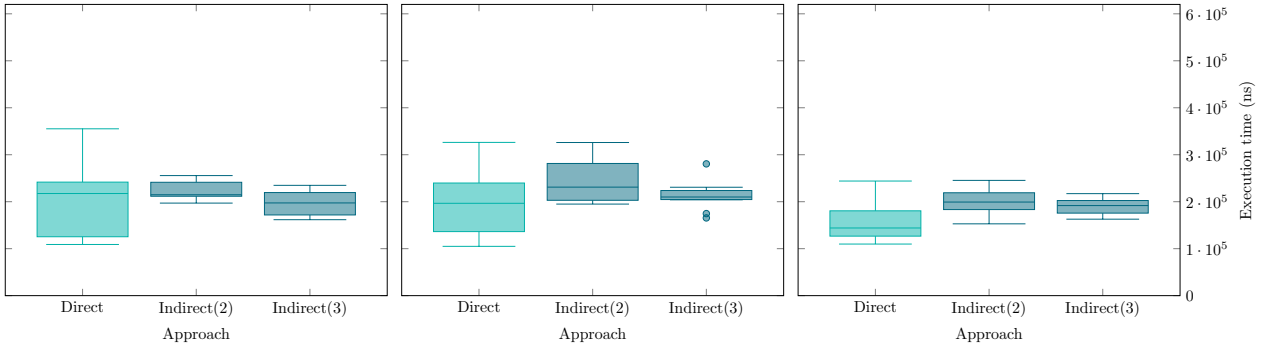
(g) Direct GA **(h)** Indirect(2) GA **(i)** Indirect(3) GA

Figure 4.6: Results of 10 trials obtained by the basic indirect approaches to the sextic equation in comparison with the direct approaches



(a) HC **(b)** SA **(c)** GA

Figure 4.7: Distribution of the best fitness values of 10 trials of the direct and basic indirect approaches on the quartic equation



(a) HC **(b)** SA **(c)** GA

Figure 4.8: Distribution of the best fitness values of 10 trials of the direct and basic indirect approaches on the sextic equation

Table 4.3: Summary of the highest fitness value of 10 trials of the direct and basic indirect approaches over different input arguments for the polynomial solver

(a) $n = 5$

Algorithm	**Direct**	**Indirect(2)**	**Indirect(3)**
HC	134,449	134,300	199,659†
SA	171,629.5	136,841	160,457
GA	75,917*	163,110	157,159

(b) $n = 7$

Algorithm	**Direct**	**Indirect(2)**	**Indirect(3)**
HC	355,198†	255,600	234,817
SA	326,249	326,003	280,537
GA	243,980	245,364	217,162*

* The worst performer. † The best performer.

Research Question 2 For the second research question, we compared the effectiveness of metaheuristic algorithms on supporting the dependent input sampling strategies to find the values of the coefficients that maximise the execution time of the polynomial solver. Figures 4.5 and 4.6 illustrate that the initial fitness values from both direct and indirect GA were higher than those from both direct and indirect single-solution based metaheuristics, i.e. HC and SA, but in the end, HC and SA outperformed in almost all cases, except the case of indirect GA with two sub-ranges on the quartic equation, where the extreme execution time appeared on the ninth trial (Figure 4.5h).

Research Question 3 Finally, the third research question is answered by comparing the results between two sets of parameter settings for the indirect approach, i.e. with two and three sub-ranges per test input argument, respectively. In four out of six cases, as depicted in Figures 4.7 and 4.8, the dependent input sampling strategy with a union of two sub-ranges gave the higher results. Only in the cases of the quartic equation with HC and SA (as shown in Figures 4.7a and 4.7b) three sub-ranges brought the extreme execution times and HC, in particular, gave the most extreme execution time for the quartic equation problem in this experiment, whereas the direct HC gave the most extreme one for the sextic equation problem as marked by † in Table 4.3.

4.5 Experiment IV—Fixed Delta-Based Interval

4.5.1 Objective

In the prior experiment, we have found that the dependent input sampling strategy with a basic interval could not surpass the direct metaheuristic approaches in some cases. The range of interval of each sub-range is varied, depending on its lower and upper bounds given by a metaheuristic algorithm.

In this experiment, we explore the impact of restricting all sub-ranges' intervals to have the same width, i.e. a fixed delta-based interval. In particular, the metaheuristic technique assigns only weight and lower bound to each node, and its upper bound is determined by adding its lower bound with a fixed delta value. This indirect method allows more exploitation on subdomain spaces. The problem statement in this experiment is also the same as in the previous experiment, that is:

Problem Statement: *For a polynomial equation of the form* $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$*, we seek values of the coefficients (*a_0 *to* a_n*) that maximise the execution time of the polynomial solver.*

In this experiment, the research questions addressed are:

Research Question 1: *Is the dependent input sampling strategy with a fixed delta-based interval effective in seeking the values of the coefficients that maximise the execution time of the polynomial solver?*

Research Question 2: *Among the metaheuristics, which technique is the best for supporting the dependent input sampling strategy with a fixed delta-based interval to seek test inputs that maximise the execution time of the polynomial solver?*

Research Question 3: *Does the number of interval choices affect the effectiveness of the dependent input sampling strategy with a fixed delta-based interval?*

4.5.2 Preparation

In this experiment, all algorithm parameter settings are almost similar to Experiment III (as summarised in Table 3.2), except that the `genome-size` given to the ECJ's metaheuristic algorithms as only a selection weight and lower bound are required to be decoded to a node of the tree structure, and an upper bound equals to the lower bound plus a delta

value. Hence, the size of a solution genome in this experiment is double the size of N. In Figure 4.9, for example, the genome size for the indirect approach with a delta-based interval of the tree-parameter (A, B, C) problem is 28.

Again, in this experiment, the polynomial function was executed with numbers of input arguments of 5 and 7, together with two options of the number of sub-ranges. Therefore, the genome sizes are summarised in Table 4.4. In addition, the delta value is given at 10.

4.5.3 Method

Four empirical studies were performed in the same way as in Experiment III, i.e. ten experimental trials were executed for each technique. Table 4.5 summarises an initial seed for each metaheuristic technique.

4.5.4 Results

For ease of reference, we use the same prefixes as defined in Experiment III, along with Δ, which indicates that it is the dependent input sampling strategy with a fixed delta-based interval. All relevant results of this experiment are shown in Figures 4.10 to 4.13 and Table 4.6.

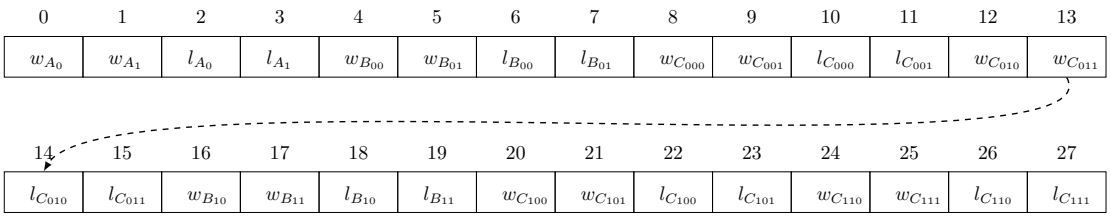


Figure 4.9: Example of a solution genome of the dependent input sampling approach with a delta-based interval

Table 4.4: Genome sizes for the empirical experiments of the dependent input sampling approach with a delta-based interval

Arguments	Sub-ranges	Genome size
5	2	124
	3	726
7	2	508
	3	6,558

Table 4.5: Initial seeds of metaheuristic algorithms for different input arguments of the polynomial root-finder (Experiment IV)

(a) 2 sub-ranges

	Arguments	
Algorithm	**5**	**7**
HC	8,378	17,928
SA	20,608	5,234
GA	−16,478	−5,291

(b) 3 sub-ranges

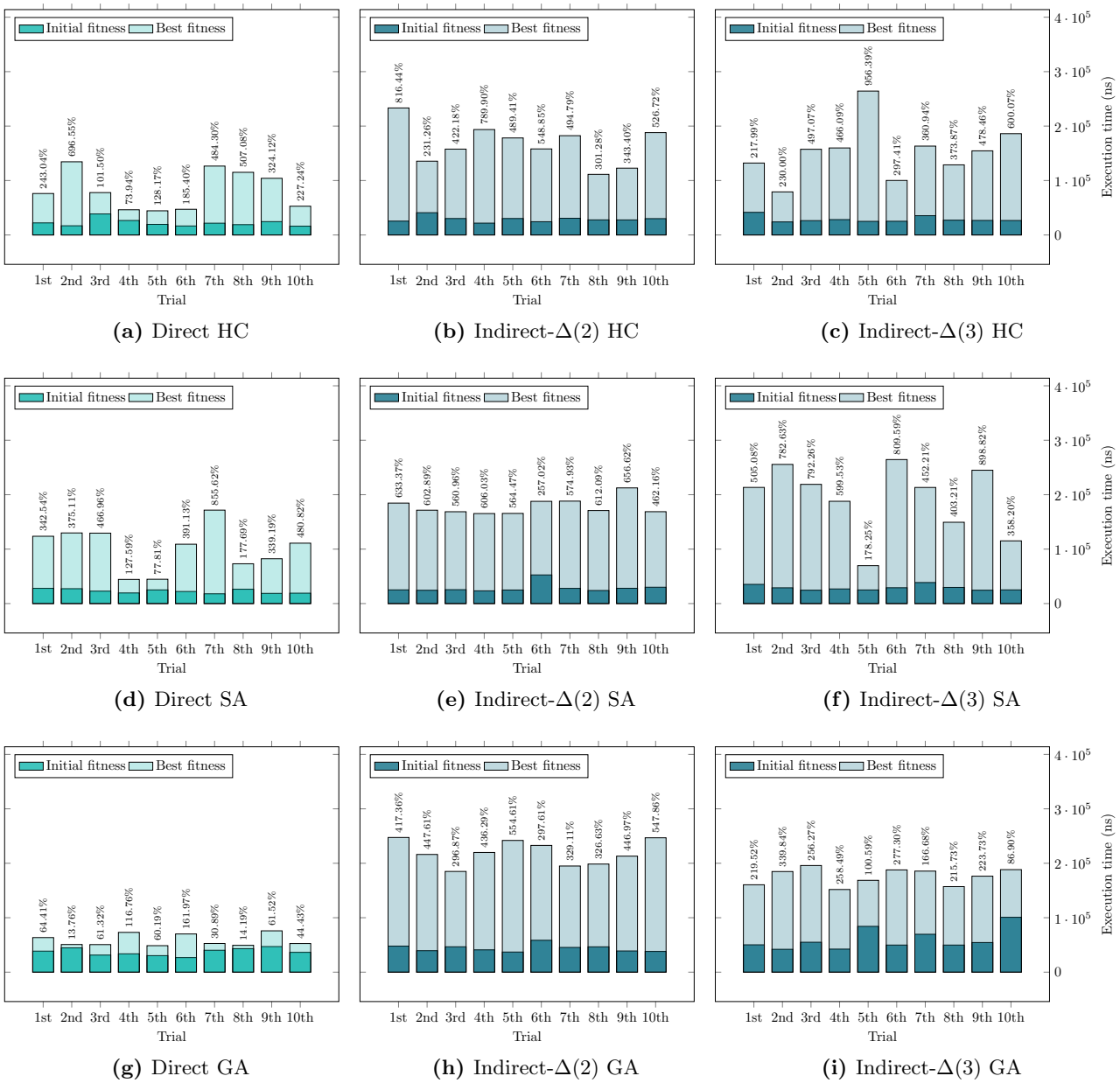
	Arguments	
Algorithm	**5**	**7**
HC	26,831	−17,756
SA	−32,118	−31,926
GA	−25,092	−26,631

4.5.5 Discussion and Conclusions

Research Question 1 For the first research question, we found that the dependent input sampling strategy with a fixed delta-based interval (where $\Delta = 10$) was superior to direct metaheuristic approaches in all cases, as illustrated in Figures 4.12 and 4.13. The results prove our aforementioned assumption in Section 4.1.2 that extreme execution times may be concentrated in small partitions of the input space and reducing the input domain could increase the chances of inputs with extreme times being sampled.

Research Question 2 In Figures 4.10 and 4.11, the bar charts demonstrate that GA started with higher initial fitness values as usual. Also, although its final fitness values were less than both HC and SA in all cases, its results with the fixed delta-based interval of indirect approach were more improved than those of direct GA and basic indirect GA. For the second research question, it can be concluded that single-solution based metaheuristics were still the most effective approaches for dependent input sampling strategies, including basic and fixed delta-based intervals.

Research Question 3 In this experiment, as illustrated in Figures 4.12 and 4.13, the fixed delta-based indirect approach with a union of three sub-ranges delivered the higher fitness values in all cases of single-solution metaheuristics, while the approach with two sub-ranges gave the higher values in cases of population-based metaheuristic. In addition, the fixed delta based interval with three sub-ranges and SA produced the most extreme execution times in this experiment, as marked by † in Table 4.6, and the most extreme execution time over all three experiments (Experiments III to V) in this chapter for the case of the quartic equation.

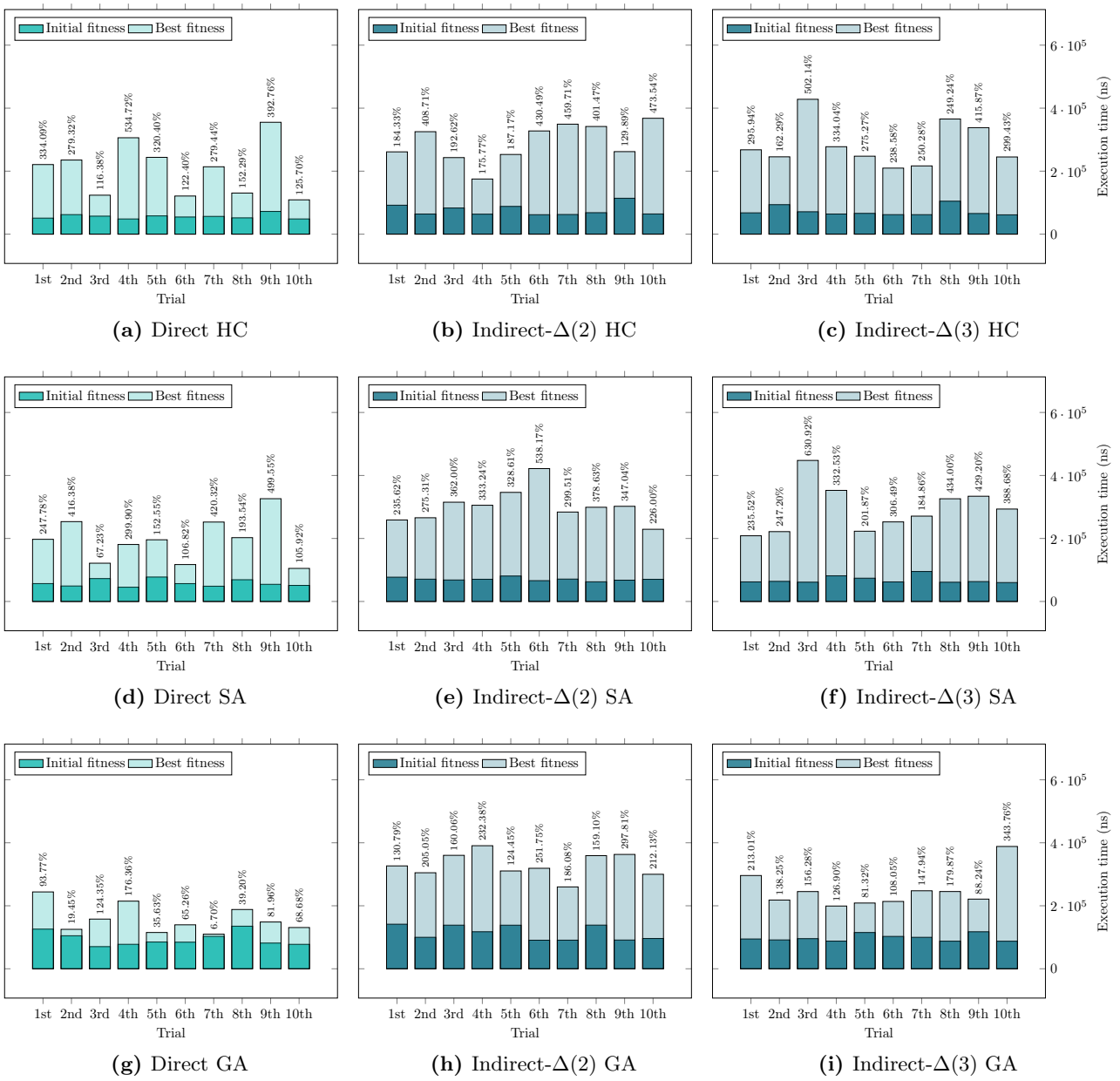


(a) Direct HC **(b)** Indirect-$\Delta(2)$ HC **(c)** Indirect-$\Delta(3)$ HC

(d) Direct SA **(e)** Indirect-$\Delta(2)$ SA **(f)** Indirect-$\Delta(3)$ SA

(g) Direct GA **(h)** Indirect-$\Delta(2)$ GA **(i)** Indirect-$\Delta(3)$ GA

Figure 4.10: Results of 10 trials obtained by the fixed delta-based indirect approaches to the quartic equation in comparison with the direct approaches

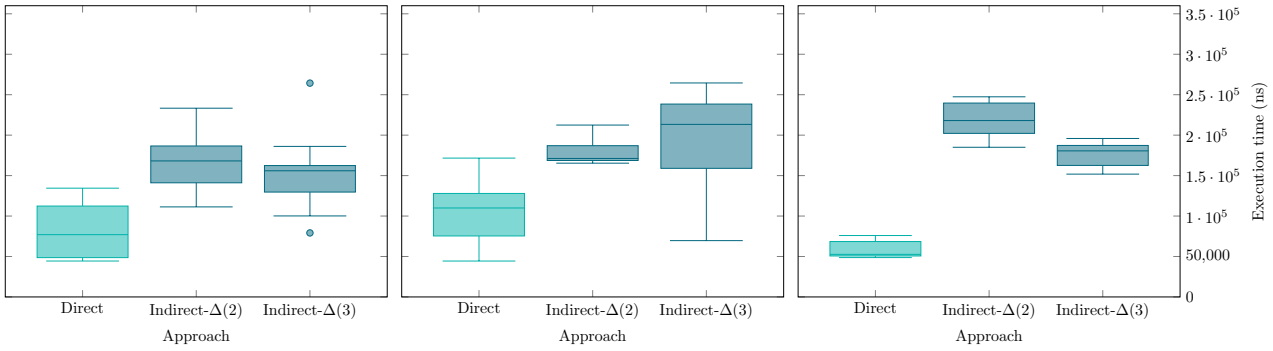


(a) Direct HC **(b)** Indirect-Δ(2) HC **(c)** Indirect-Δ(3) HC

(d) Direct SA **(e)** Indirect-Δ(2) SA **(f)** Indirect-Δ(3) SA

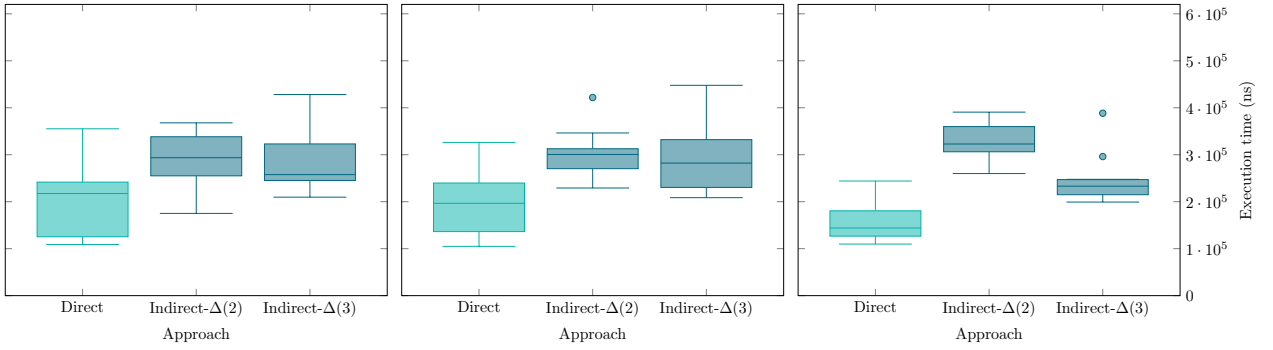
(g) Direct GA **(h)** Indirect-Δ(2) GA **(i)** Indirect-Δ(3) GA

Figure 4.11: Results of 10 trials obtained by the fixed delta-based indirect approaches to the sextic equation in comparison with the direct approaches



(a) HC **(b)** SA **(c)** GA

Figure 4.12: Distribution of the best fitness values of 10 trials of the direct and fixed delta-based indirect approaches on the quartic equation



(a) HC **(b)** SA **(c)** GA

Figure 4.13: Distribution of the best fitness values of 10 trials of the direct and fixed delta-based indirect approaches on the sextic equation

Table 4.6: Summary of the highest fitness value of 10 trials of the direct and fixed delta-based indirect approaches over different input arguments for the polynomial solver

(a) $n = 5$

Algorithm	**Direct**	**Indirect-Δ(2)**	**Indirect-Δ(3)**
HC	134,449	233,315.5	264,310
SA	171,629.5	212,466	264,523†
GA	75,917*	247,400	195,888

(b) $n = 7$

Algorithm	**Direct**	**Indirect-Δ(2)**	**Indirect-Δ(3)**
HC	355,198	367,868	428,145
SA	326,249	346,301	447,760†
GA	243,980*	390,689.5	388,320

* The worst performer. † The best performer.

4.6 Experiment V—Randomised Delta-Based Interval

4.6.1 Objective

The restriction of the interval ranges of sub-ranges with a fixed delta value was found effective to support the dependent input sampling strategies in seeking values of coefficients of the quartic and sextic equations as evidenced in Experiment IV. In this experiment, we make it more flexible, but still under control, by sampling a delta value for each sub-range from a given range, e.g. $[1, 20]$. This randomised delta-based indirect method allows a bit more exploration on subdomain spaces. The problem statement in this experiment is also the same to that of Experiment III, which is:

Problem Statement: *For a polynomial equation of the form* $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$*, we seek values of the coefficients (*a_0 *to* a_n*) that maximise the execution time of the polynomial solver.*

The research questions addressed in this experiment are:

Research Question 1: *Is the dependent input sampling strategy with a randomised delta-based interval effective in seeking the values of the coefficients that maximise the execution time of the polynomial solver?*

Research Question 2: *Among the metaheuristics, which technique is the best for supporting the dependent input sampling strategy with a randomised delta-based interval to seek test inputs that maximise the execution time of the polynomial solver?*

Research Question 3: *Does the number of interval choices affect the effectiveness of the dependent input sampling strategy with a randomised delta-based interval?*

4.6.2 Preparation

All algorithm parameter settings and the genome sizes in this experiment are also the same as in the preceding experiment (Experiment IV), which are shown in Tables 3.2 and 4.4, respectively. However, a randomised delta value is randomly sampled from a uniform distribution $U(1, 20)$.

4.6.3 Method

Each metaheuristic technique was executed in combination with the randomised delta based method for ten experimental trials. An initial seed obtained from `random.org` for each technique is summarised in Table 4.7.

4.6.4 Results

For ease of reference, the same prefixes defined in Experiment III are also used, together with Δ_r, which indicates that it is the dependent input sampling strategy with a randomised delta-based interval. The results of this experiment are illustrated in Figures 4.14 to 4.17, and Table 4.8.

4.6.5 Discussion and Conclusions

Research Question 1 This more flexible version of the fixed delta-based indirect approach also outperformed the direct metaheuristics in all cases, as depicted in Figures 4.16 and 4.17. This, again, confirmed our assumption stated in Section 4.1.2.

Research Question 2 In Table 4.8, the results show that for two sub-ranges of the randomised delta-based interval of the indirect approach, GA outperformed SA in both quartic and sextic equation problems. Also, in case of a union of three sub-ranges in the quartic equation, GA surpassed HC. However, overall, SA and HC still produced the most extreme execution times on quartic and sextic equations, respectively, in this experiment.

Research Question 3 Furthermore, the three sub-ranges were better than the two sub-ranges in cases of SA and GA of the quartic equation problem, as shown in Figures 4.16b and 4.16c.

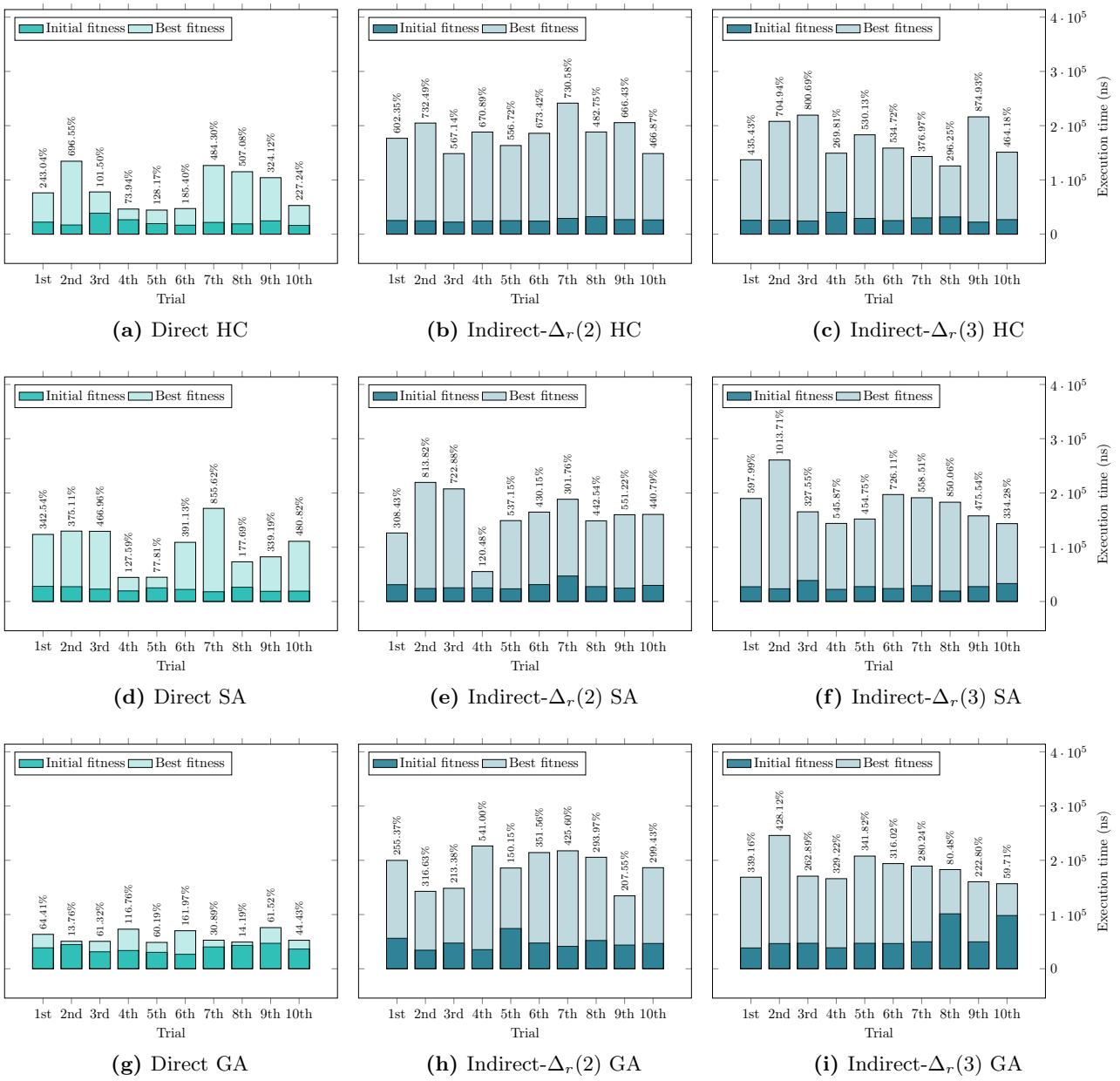
Table 4.7: Initial seeds of metaheuristic algorithms for different input arguments of the polynomial root-finder (Experiment V)

(a) 2 sub-ranges

	Arguments	
Algorithm	**5**	**7**
HC	−31,947	19,067
SA	−31,084	−3,811
GA	−4,226	−22,853

(b) 3 sub-ranges

	Arguments	
Algorithm	**5**	**7**
HC	13,628	−13,422
SA	−29,340	8,718
GA	−22,916	−28,746



(a) Direct HC **(b)** Indirect-$\Delta_r(2)$ HC **(c)** Indirect-$\Delta_r(3)$ HC

(d) Direct SA **(e)** Indirect-$\Delta_r(2)$ SA **(f)** Indirect-$\Delta_r(3)$ SA

(g) Direct GA **(h)** Indirect-$\Delta_r(2)$ GA **(i)** Indirect-$\Delta_r(3)$ GA

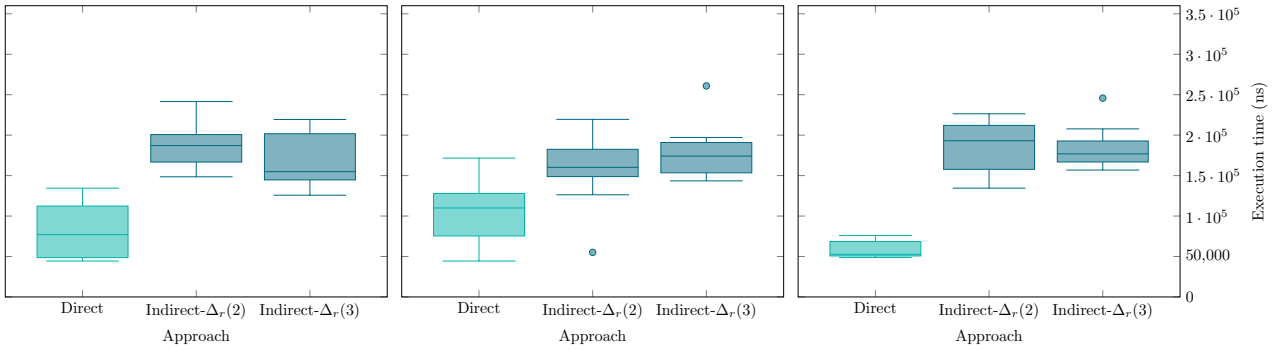
Figure 4.14: Results of 10 trials obtained by the randomised delta-based indirect approaches to the quartic equation in comparison with the direct approaches

(a) Direct HC **(b)** Indirect-$\Delta_r(2)$ HC **(c)** Indirect-$\Delta_r(3)$ HC

(d) Direct SA **(e)** Indirect-$\Delta_r(2)$ SA **(f)** Indirect-$\Delta_r(3)$ SA

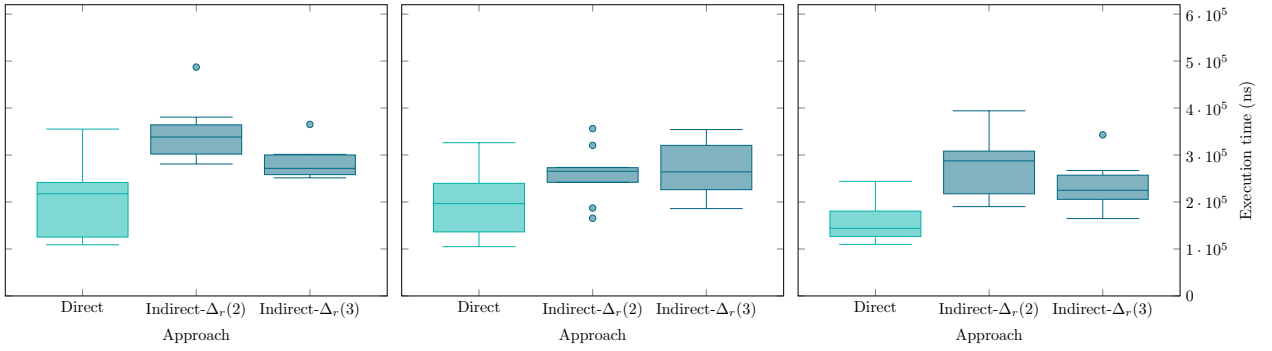
(g) Direct GA **(h)** Indirect-$\Delta_r(2)$ GA **(i)** Indirect-$\Delta_r(3)$ GA

Figure 4.15: Results of 10 trials obtained by the randomised delta-based indirect approaches to the sextic equation in comparison with the direct approaches



(a) HC **(b)** SA **(c)** GA

Figure 4.16: Distribution of the best fitness values of 10 trials of the direct and randomised delta-based indirect approaches on the quartic equation



(a) HC **(b)** SA **(c)** GA

Figure 4.17: Distribution of the best fitness values of 10 trials of the direct and randomised delta-based indirect approaches on the sextic equation

On the whole, as marked by † in Table 4.8, SA and the randomised delta-based indirect approach with a union of three sub-ranges was the best for the quartic equation problem, whereas HC and the randomised delta-based indirect approach with two sub-ranges was the best for the sextic equation problem in this experiment and also among three experiments (Experiments III to V).

Table 4.8: Summary of the highest fitness value of 10 trials of the direct and randomised delta-based indirect approaches over different input arguments for the polynomial solver

(a) $n = 5$

Algorithm	**Direct**	**Indirect-Δ_r(2)**	**Indirect-Δ_r(3)**
HC	134,449	241,541	219,380
SA	171,629.5	219,500	260,820†
GA	75,917*	226,421	245,780

(b) $n = 7$

Algorithm	**Direct**	**Indirect-Δ_r(2)**	**Indirect-Δ_r(3)**
HC	355,198	487,240†	365,160
SA	326,249	356,321	354,259.5
GA	243,980*	394,097	343,003

* The worst performer. † The best performer.

4.7 Discussion as a Whole

According to the results from the experiments in the previous three section, the indirect approach is suggested to be an effective means of revealing optimal test inputs for the temporal testing problem in comparison with the direct approach. Since the number of sampling trials per an established sampling distribution is given to ten, the indirect approach here, therefore, requires ten times as many timing measurements compared to the direct approach. However, when the direct approach, i.e. GA, takes the same timing measurements, i.e. increasing its number of generations from 101 to 1,000, as previously evidenced in the preliminary analysis of Chapter 3 (in Figure 3.11 in particular), there was almost no improvement after a long period of running GA.

Among three different types of intervals, the delta-based intervals, i.e. fixed delta-based and randomised delta-based intervals, delivered more extremal test vectors than the basic interval. The domain sub-ranges based on the basic interval are varied, depending on the lower upper bounds decoded to them. Although such sub-ranges are undoubtedly smaller than the full domain range of the direct approach, they may still be considered too large for temporal test input generation. Often, the bounds are broader than the cases of delta-based intervals, where the bounds are either fixed or random to small numbers.

Table 4.9 summarises the test input arguments that provided the most extreme execution times for the quartic and sextic polynomials over all those three experiments of the dependent input sampling strategies. The best values of coefficients for the quartic polynomial was gained by the indirect approach with a union of three sub-ranges, where their intervals are fixed to ten ($\Delta = 10$). On the other hand, the best values of coefficients for the sextic polynomial was discovered by the indirect approach with a union of two sub-ranges, where their intervals are random between the range of $[1, 20]$ ($\Delta_r = U(1, 20)$).

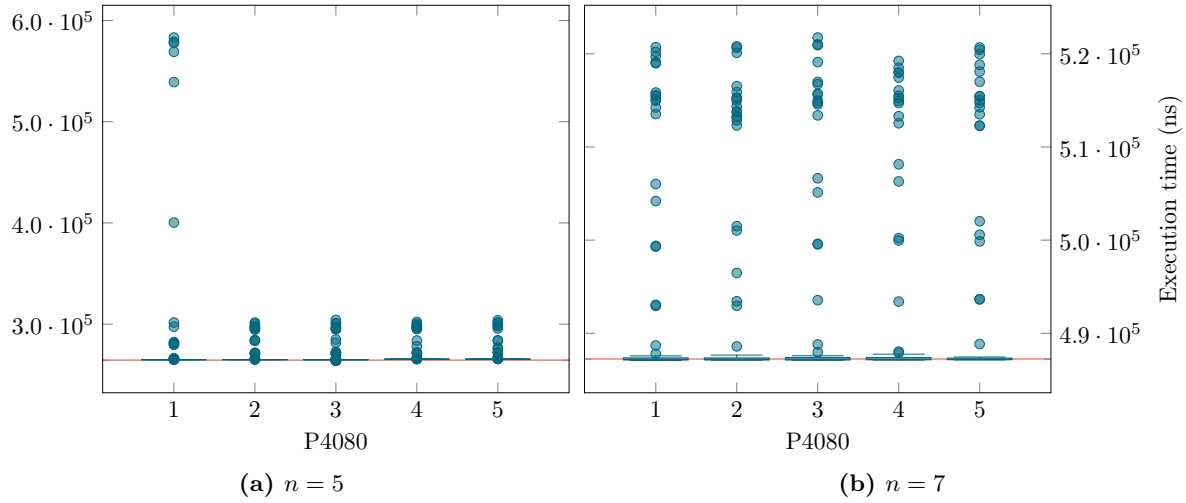
In addition, the numbers of QR iterations are also listed in Table 4.9. Compared to the best input arguments in Chapter 3, the best arguments obtained in this chapter require more iterations for convergence.

We also validated these best input arguments by rerunning them on five P4080 development boards in the same way as in Chapter 3. The execution times collected from 100 runs are plotted in Figure 4.18. The box-and-whisker plots showed that there were almost no variations over the probability distributions of 100 execution times, as indicated by the midspreads across five different boards. Also, all development boards' midspreads were at the same level and the actual execution times are within them in both cases.

Moreover, these best input arguments were sampled from the selected sampling regimes, which were constructed from the solution genomes presented in Table D.4 of Appendix D. The tree representations for sampling distribution of such solution genomes are illustrated in the following figures. Particularly, Figures 4.19 to 4.27 present the tree structure from which the best input arguments of the quartic equation were sampled, and Figures 4.28 to 4.31 present the tree structure from which the best input arguments of the sextic equation were sampled. There are two elements presented in each node of the tree, i.e. a selection weight (w) and a lower bound (l) in the form of $\frac{w}{[l]}$.

Table 4.9: Best values of coefficients (Chapter 4)

Arguments	Coefficient vector	Iterations
5	−21,492; −11,333; 26,663; 10,440 and −13,244	77
7	6,134; −16,035; 28,217; −16,510; −15,982; 32,398 and 17,391	74



(a) $n = 5$ **(b)** $n = 7$

Figure 4.18: Distribution of 100 execution times of polynomial solver with best input arguments obtained from 5 P4080 boards (Chapter 4)

According to those tree representations, the best input arguments for the quartic equation were sampled from the sampling regime, which consists of the dependent sub-ranges of $[-21498, -21488]$, $[-11340, -11330]$, $[26660, 26670]$, $[10437, 10447]$ and $[-13245, -13244]$, respectively, as highlighted in Figure 4.21.

Furthermore, as highlighted in Figure 4.31 for the sextic equation, the best input arguments were sampled from the sampling regime, which contains the dependent sub-ranges of $[6123, 6123 + U(1, 20)]$, $[-16050, -16050 + U(1, 20)]$, $[28217, 28217 + U(1, 20)]$, $[-16511, -16511 + U(1, 20)]$, $[-15982, -15982 + U(1, 20)]$, $[32396, 32396 + U(1, 20)]$ and $[17391, 17391 + U(1, 20)]$, respectively.

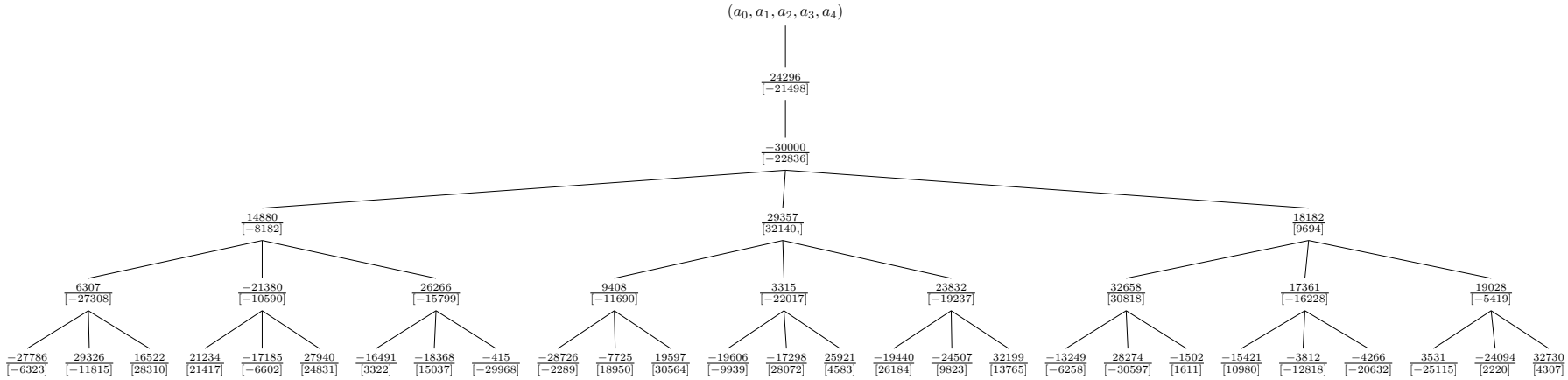


Figure 4.19: Tree structure decoded from the best solution genome for the quartic equation (1st part)

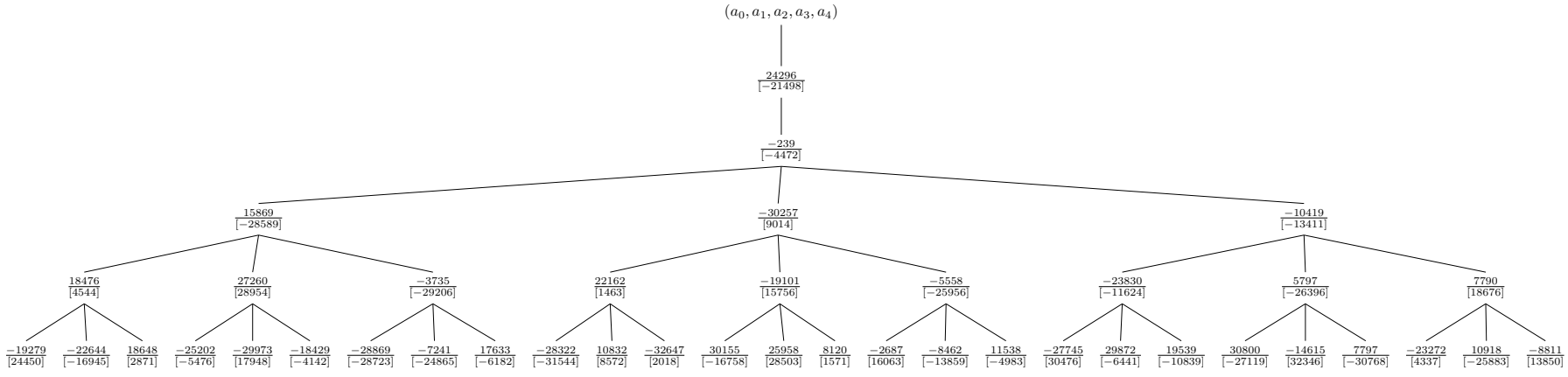


Figure 4.20: Tree structure decoded from the best solution genome for the quartic equation (2nd part)

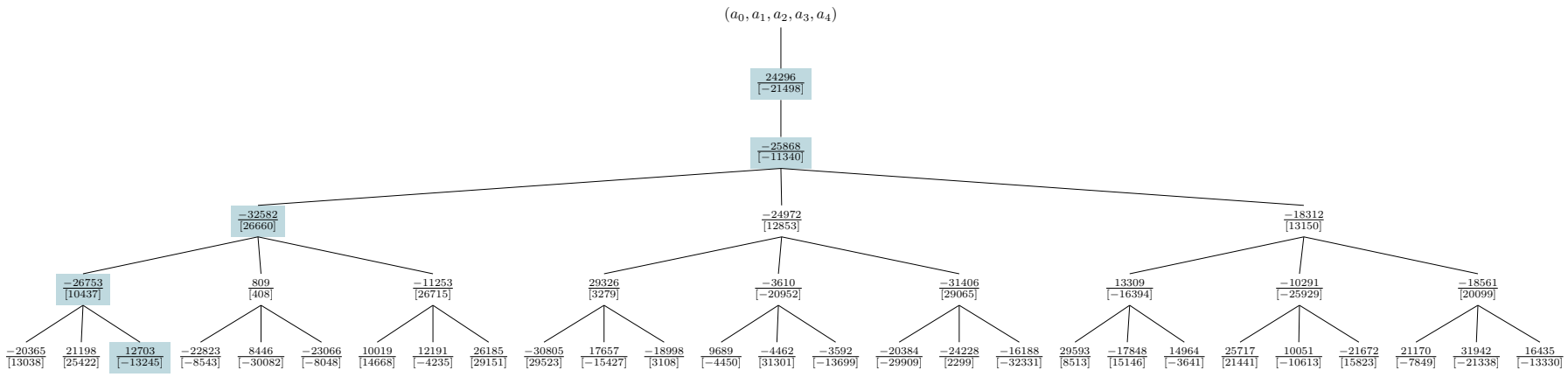


Figure 4.21: Tree structure decoded from the best solution genome for the quartic equation (3rd part)

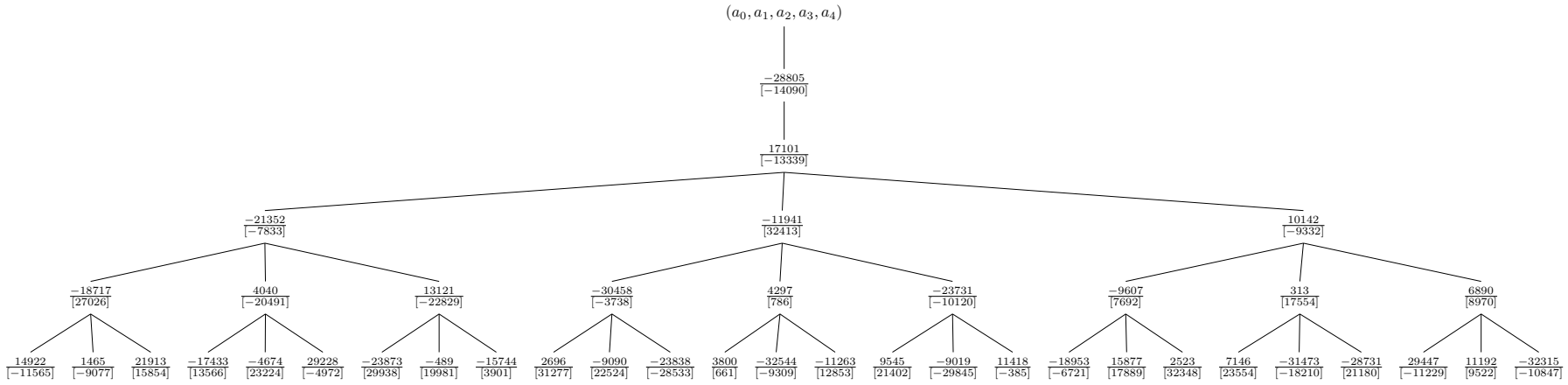


Figure 4.22: Tree structure decoded from the best solution genome for the quartic equation (4th part)

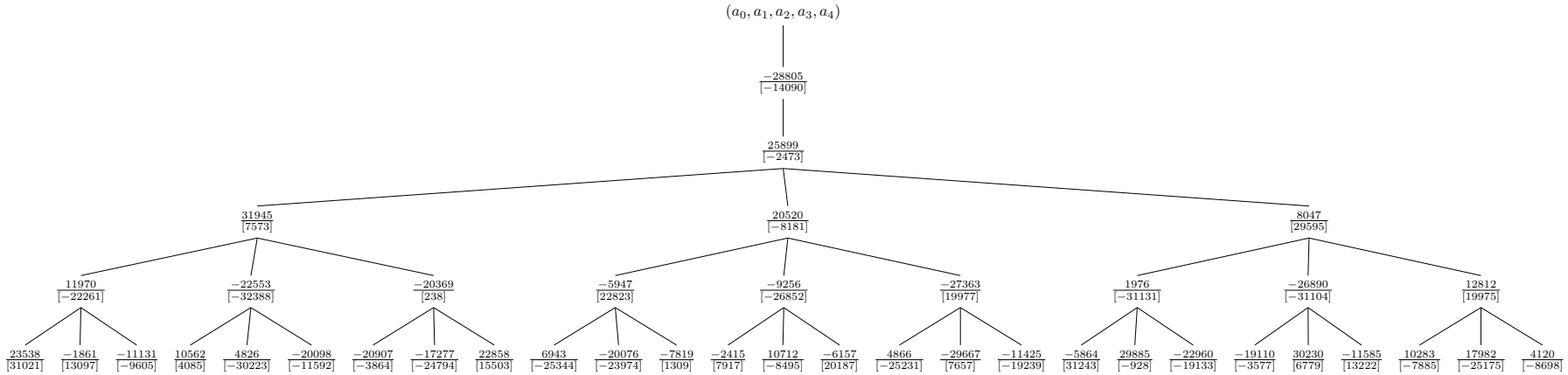


Figure 4.23: Tree structure decoded from the best solution genome for the quartic equation (5th part)

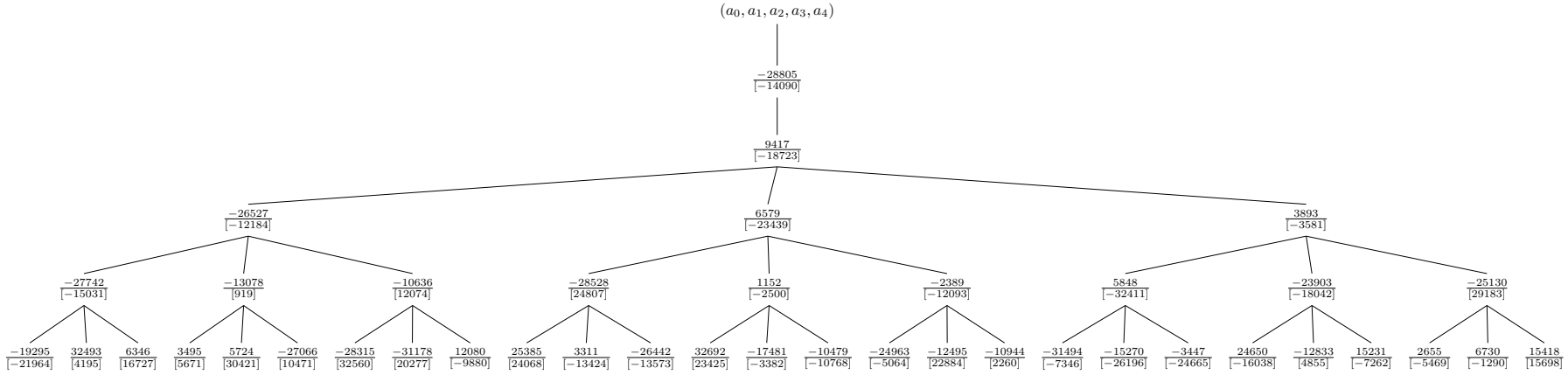


Figure 4.24: Tree structure decoded from the best solution genome for the quartic equation (6th part)

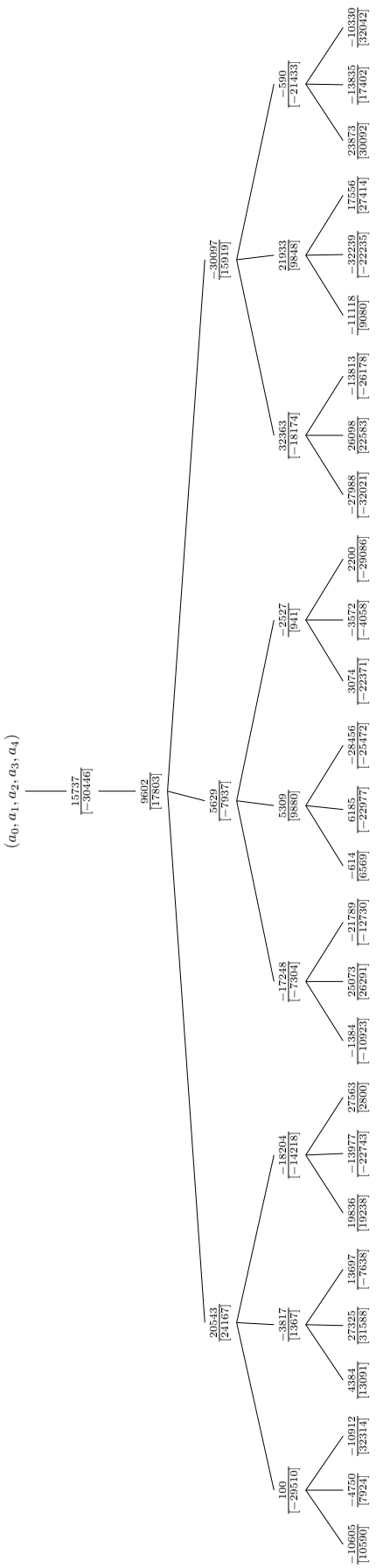


Figure 4.25: Tree structure decoded from the best solution genome for the quartic equation (7th part)

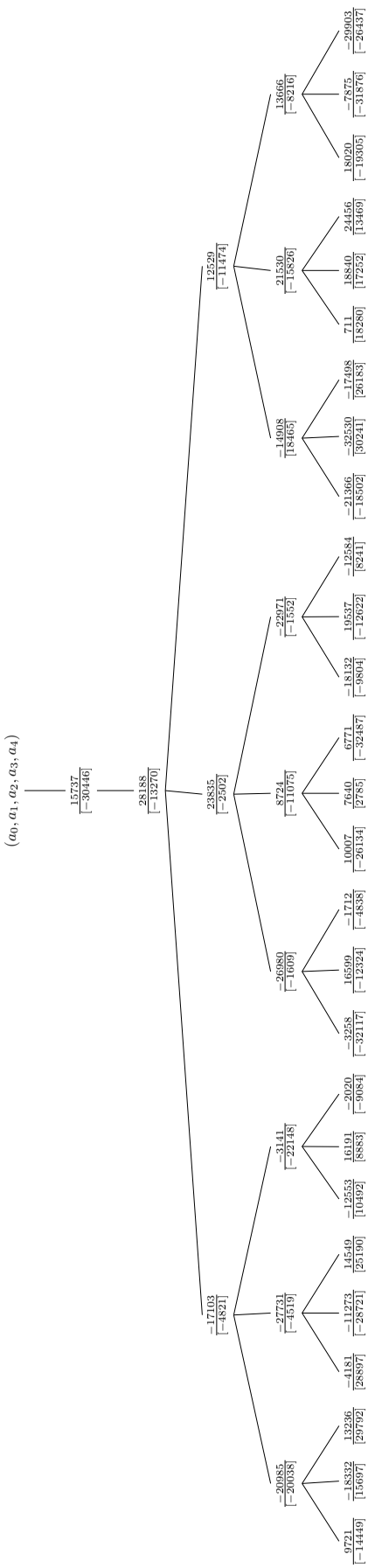


Figure 4.26: Tree structure decoded from the best solution genome for the quartic equation (8th part)

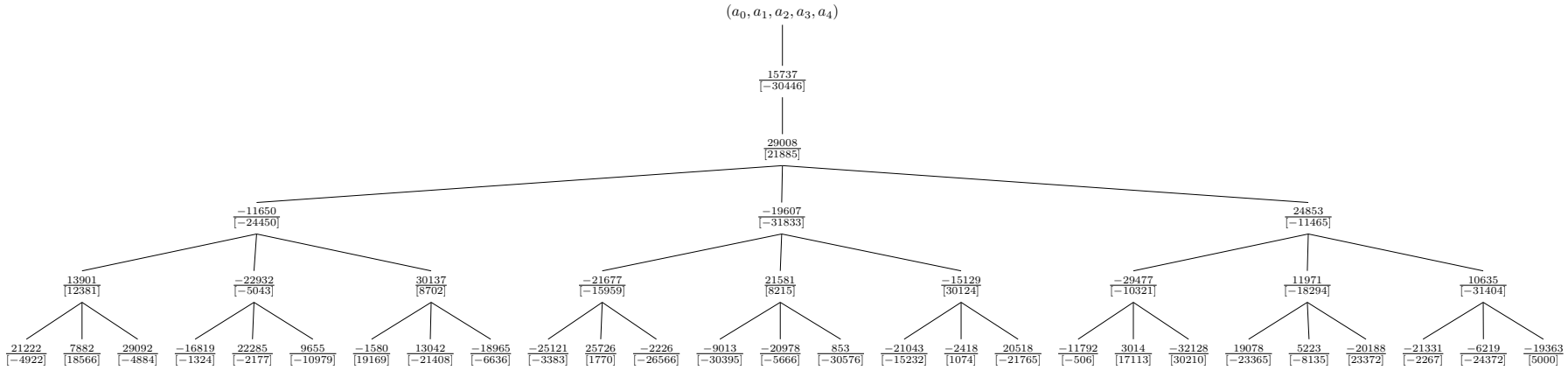


Figure 4.27: Tree structure decoded from the best solution genome for the quartic equation (9th part)

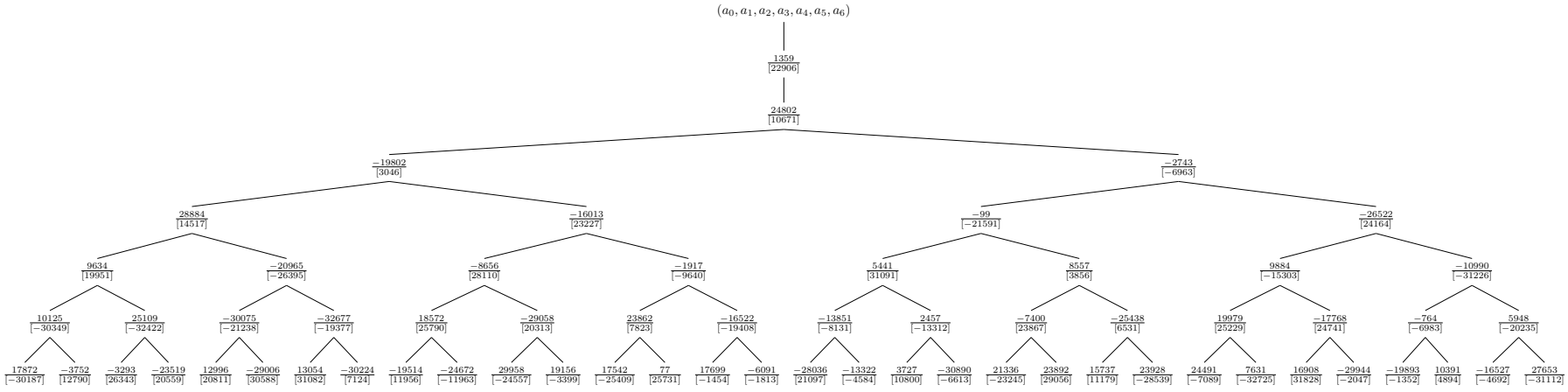


Figure 4.28: Tree structure decoded from the best solution genome for the sextic equation (1st part)

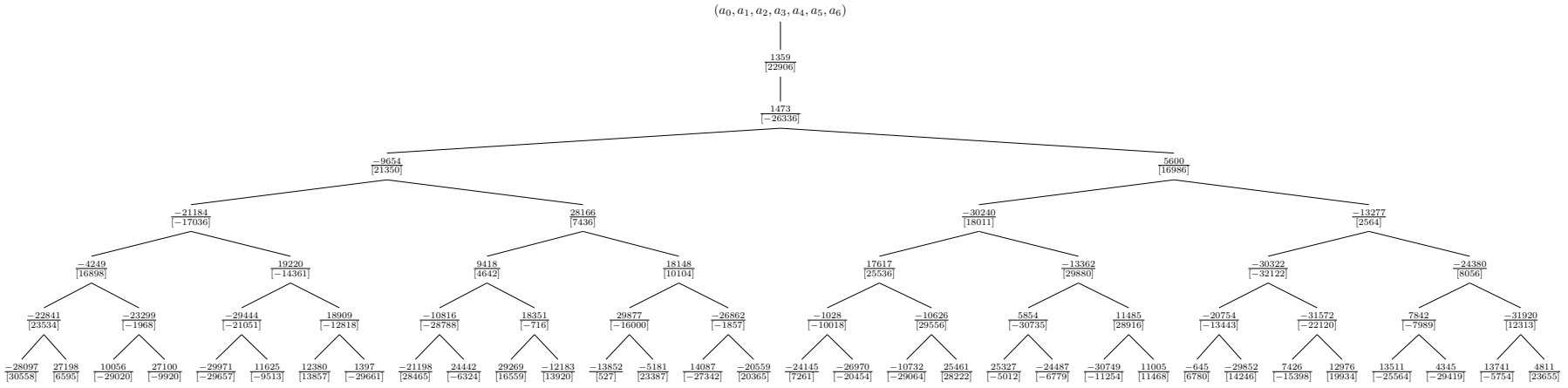


Figure 4.29: Tree structure decoded from the best solution genome for the sextic equation (2nd part)

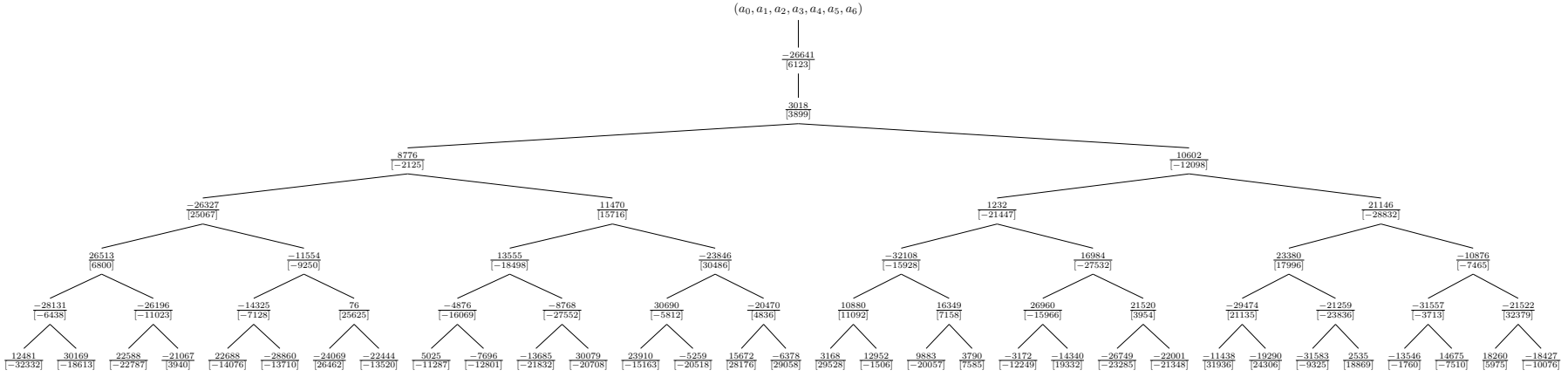


Figure 4.30: Tree structure decoded from the best solution genome for the sextic equation (3rd part)

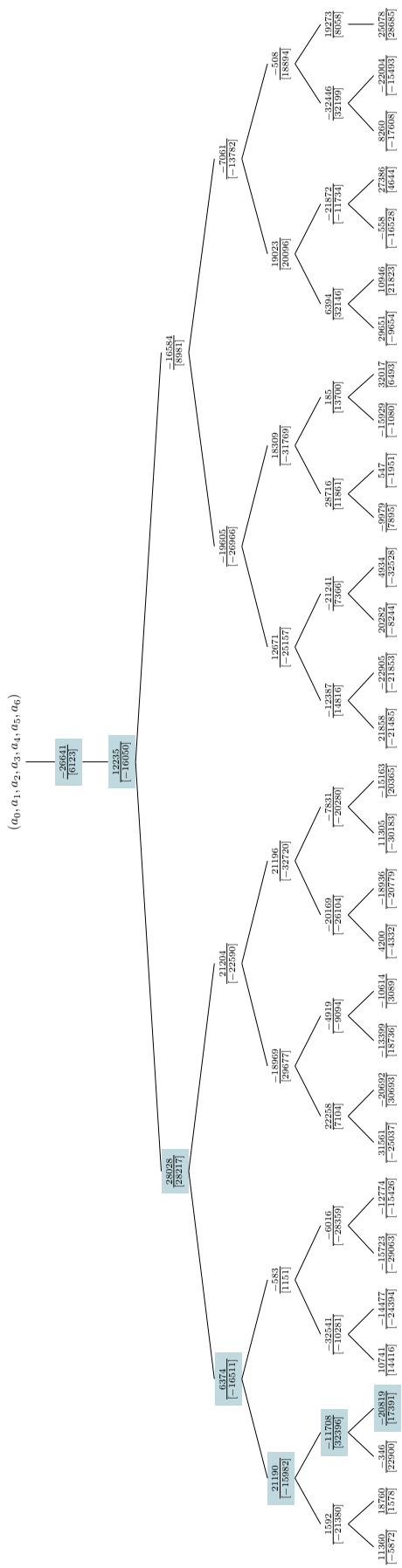


Figure 4.31: Tree structure decoded from the best solution genome for the sextic equation (4th part)

4.8 Summary

To sum up, this chapter investigated the ability of our proposed approach, i.e. dependent input sampling strategies, to seek temporal input arguments that maximise the execution time of a task running on a multicore environment. The proposed approach is based on the assumption that the test inputs that maximise the execution time may be concentrated in small parts of the input space, and also such inputs may have a correlation among themselves that escalate the execution time. In order to increase the chances of test inputs with extreme times being sampled, we present here the novel approach that aims to reduce the whole input domain into a subdomain, which is a union of a number of domain sub-ranges. Also, the approach samples a parameter from the probability-based selective interval, which depends on the specific intervals sampled from previous parameters.

Three essential parameters, i.e. the number of sub-ranges, the type of interval and the number of sampling trials, trigger the approach to balance between the exploration and exploitation over the domain input space. In particular, the first parameter diverges the search over the input space, whereas the second parameter intensifies on a small partition on such the space. The last parameter enables the approach to repeatedly sample a number of test vectors from the established sampling distribution.

The experiments in this chapter were separated into three sub-experiments (Experiments III to V) based on the different types of intervals: 1) a basic interval; 2) a fixed delta-based interval; and 3) a randomised delta-based interval. In each sub-experiment, the number of sampling trials was given at ten, whereas the number of sub-ranges was two and three.

Overall, the results from three empirical experiments demonstrate that our dependent approach performed effectively when the intervals of sub-ranges are restricted with either a fixed delta value ($\Delta = 10$) or a randomised value of delta from a given range ($\Delta_r = U(1, 20)$). However, the direct approach outperformed the indirect approach with a basic interval in several cases. According to the results, it could be explained that the interval of the basic case is varied based on its given lower and upper bounds. Such interval sometimes may be too wide compared to those of the delta cases, where the ranges are small. As a result, the exploitability of the indirect approach is limited.

Furthermore, the elements value, i.e. a lower bound, an upper bound and a solution weight, of sub-ranges created from the dependent input sampling strategies are given by a metaheuristic optimisation method. Therefore, we also explored the ability of the metaheuristic methods to evolve the values of parameters assigned to such subdomains of the sampling distribution. The search space for these optimisation techniques is, therefore, the space of parameterised distributions; not the space of inputs. Regarding the results from those three experiments, single-solution based metaheuristics, i.e. HC and SA, were effective in supporting the indirect approach to finding test vectors that maximise the execution time of the polynomial solver.

As aforementioned in Section 4.2.1, the metaheuristic optimisation techniques were used in this chapter to search on a parameterised distribution space, and the empirical results demonstrate that using them to explore the space of parameterised distributions is more effective for the problem domain of temporal testing than using them to search a space of inputs directly. In the subsequent chapter, we will investigate the effectiveness of a hyper-heuristic—a heuristic search method that operates on a search space of heuristics—on verifying temporal constraints.

Chapter 5

Hyper-Heuristics

5.1 Introduction

5.1.1 Motivation

The results from the previous two chapters have shown that metaheuristic approaches, especially HC and SA, are effective in solving the problem of temporal testing in an embedded multicore environment. The research has investigated the use of optimisation techniques directly on the test input space and indirectly, where the target of optimisation is the space of dependent randomised input strategies.

However, it is far from clear, a priori, that using any specific optimisation technique alone will give the best results. Different optimisation approaches may work effectively in different parts of the same search space. It seems plausible to see how extant techniques can be combined in some way. However, it should not be assumed that the human tester is capable of determining which way this might most profitably be done. This general problem is known to the optimisation community and has led to the development of higher-level heuristic search methods, termed 'hyper-heuristics'. These are now acknowledged by many as a more generic effective means of solving real-world computational problems [60].

This 'off-the-peg' approach operates on a search space of *heuristics* (or heuristic components) to *select* or *generate* heuristics, and an adequate combination of the selected heuristics (or generated components) is applied to solve the underlying problem [60]. Its generality has been claimed to allow easy application to newly encountered problems (or even new instances of similar problems) [60].

The literature [60] shows the success of hyper-heuristics on diverse application domains such as production scheduling, personnel scheduling, educational timetabling, cutting and packing, workforce scheduling, constraint satisfaction, vehicle routing and travel salesman problem.

In this chapter, we examine the effectiveness of using a hyper-heuristic to find test cases exhibiting extreme execution times of a task running on an embedded multicore platform. We believe this is the first application of hyper-heuristics to this problem.

5.1.2 Contribution

The contribution in this chapter is:

- The provision of empirical evidence to demonstrate that an EA hyper-heuristic is an effective way of reaching extreme execution times of numerical functions running on an embedded multicore chip but is inefficient in terms of computational demands.

5.1.3 Chapter Outline

The remainder of this chapter is organised as follows:

Section 5.2 briefly describes a hyper-heuristic toolkit, as well as a number of defined low-level heuristics, employed in this chapter.

In Section 5.3, a preliminary analysis is carried out to simplify how the particular parameter values are set for the hyper-heuristic toolkit.

Experiment VI (Section 5.4) describes and reports the results of an empirical study, i.e. applying a hyper-heuristic approach for extreme timing performance of a (single-threaded) numerical function.

Finally, the preliminary results gained from the experiment are discussed in Section 5.5.

5.2 Hyper-Heuristic Toolkit

The empirical experiments in this chapter were similarly conducted using the experimental framework and procedure described previously in Chapter 3 (Section 3.2). However, since there are no hyper-heuristic features available in ECJ, the experiments in this chapter were instead facilitated by EvoHyp [125], which is a Java-based toolkit for EA hyper-heuristics. In particular, the toolkit uses GA as a high-level algorithm to choose some low-level perturbative heuristics or search operators.

In fact, other Java-based hyper-heuristic toolkits, such as HyFlex [126] and HYPERION [127], are also publically available. Nevertheless, this thesis rather aims at applying the hyper-heuristic to the temporal testing problem. We neither intend to develop a new iterative general-purpose heuristic search algorithm (also called a 'hyper-heuristic'), which is the purpose of HyFlex [126] nor to generate new hyper-heuristics by using metaheuristics, which is the aim of HYPERION [127].

In general, EvoHyp uses generational evolution, where a whole population is evaluated and then updated at a time. The evolutionary mechanism of EvoHyp is illustrated in Figure 5.1. In particular, each individual or chromosome is a combination of low-level heuristics, which are represented by a string of characters, such as a, b, c, d, e, f and g.

As shown in the upper dark (sky blue) ellipse area of Figure 5.1, which represents the population of hyper-heuristics, its heuristic individuals, i.e. chromosomes 1, 2 and n, are initially created at random. For example, chromosome 1 is randomly generated to be a combination of the low-level heuristics e, a and b.

Within a chromosome, its initial solution, which is a test input in general and a temporal test vector in particular, is randomly generated. Then, the perturbative heuristics of such chromosome will be applied to the initial candidate solution in turn, from left to right [128], as illustrated in the lower (light green) ellipse area of Figure 5.1, which represents the space of test inputs. In other words, such initial solution will be perturbed by a set of randomly selected perturbative heuristics and will then be evaluated its fitness function, which represents a quality of the individual.

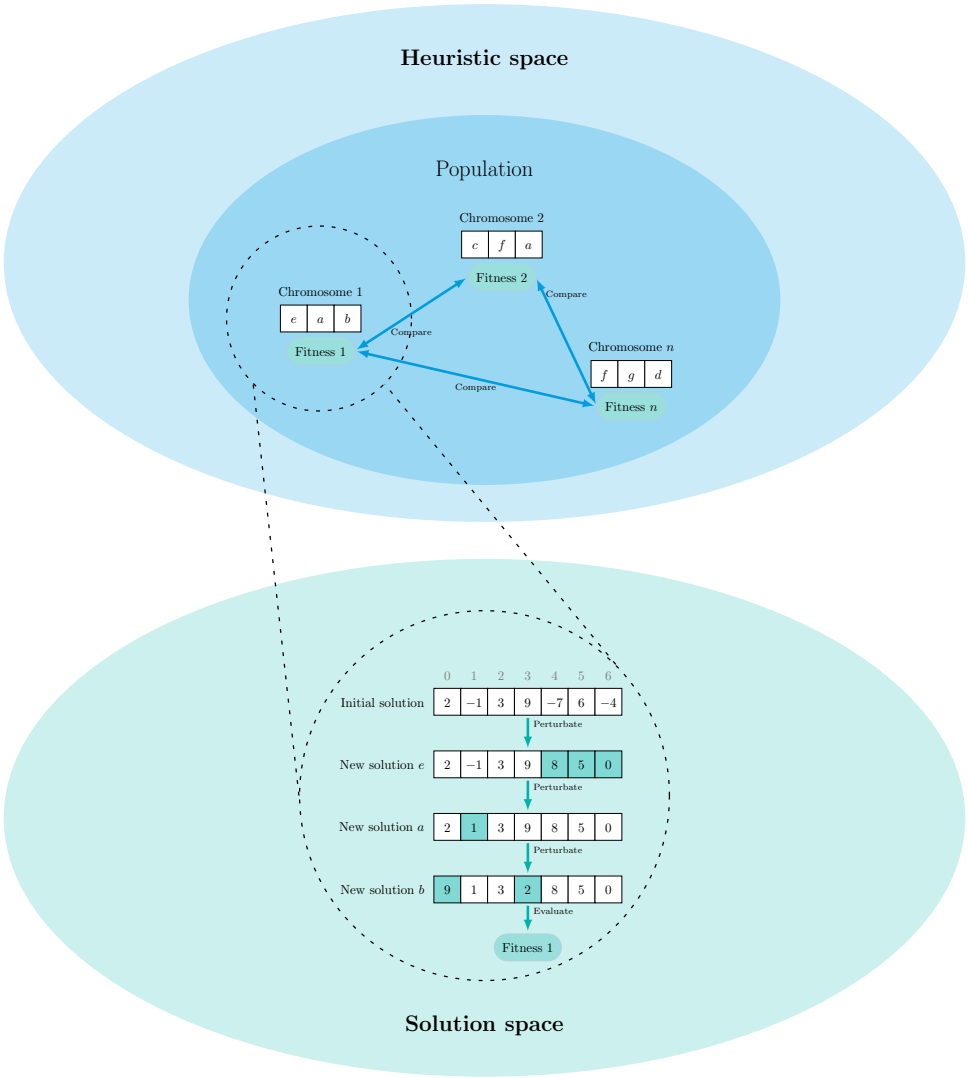


Figure 5.1: Overview of EA-based hyper-heuristics provided by EvoHyp toolkit

Overall, since the toolkit employs GA as a high-level mechanism for operating low-level heuristics, therefore, EvoHyp creates an initial population and evaluates the fitness values of chromosomes within such initial population [128]. EvoHyp then selects parents and followed by applies genetic operations to produce the offspring for the next generation of the population [128]. After that, in each generation, the process iteratively evaluates its new population until the termination criterion is satisfied [128].

In addition, in order to utilise EvoHyp toolkit for temporal testing, low-level perturbative heuristics should be defined and implemented. Accordingly, we defined seven low-level heuristics for temporal test input generation. The descriptions of such newly defined heuristics are presented in the following subsection.

5.2.1 Low-Level Heuristics

In the chapter, we defined low-level perturbative heuristic choices for the toolkit by adapting the definitions from mutators of bit-vector representation presented in [57]. Moreover, we classified the newly defined low-level heuristics into three categories based on the existing one presented in [126], which includes mutation, ruin and recreate, and local search. Below are the descriptions of the proposed heuristics:

Mutation Mutation is a heuristic which randomly mutates a solution [126]. This may return a solution which is worse than the original one.

Converting Converting mutator randomly selects positions of the vector based on a mutation probability, and then converts the values on those selected positions from a positive to a negative number, and vice versa, as shown in Figure 5.2.

Interchanging Interchanging mutator randomly selects two positions of the vector, and then the values corresponding to these two positions are interchanged, as shown in Figure 5.3.

Reversing Revering mutator randomly selects a position of the vector, and then the values after that position are reversed, as shown in Figure 5.4.

Ruin and Recreate Based on [126], the ruin and recreate category is originally defined as a perturbative heuristic which changes part of a solution and then attempts to recreate or repair it. In this research, however, the newly defined low-level heuristics do not have a 'recreate/repair' stage.

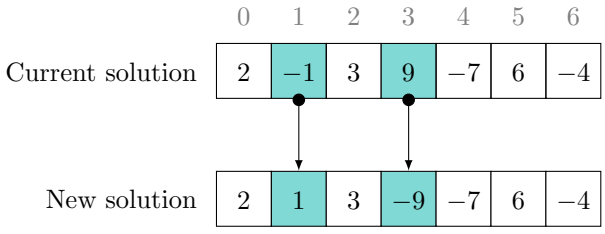


Figure 5.2: Mutation converting

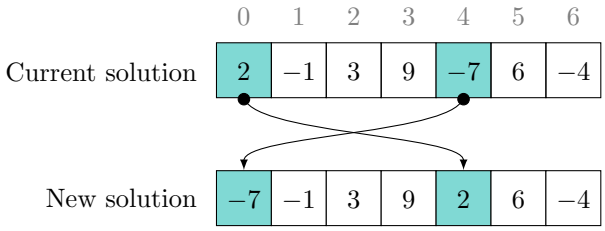


Figure 5.3: Mutation interchanging

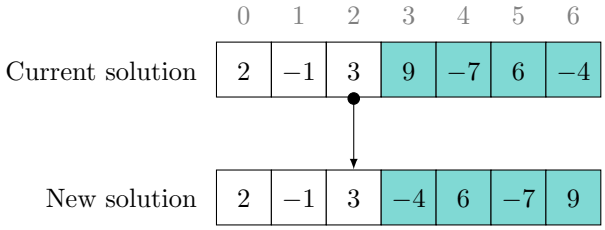


Figure 5.4: Mutation reversing

Tweaking Tweaking operator randomly selects positions of the vector, and then randomly generates new values to those selected positions, as shown in Figure 5.5.

Inserting Inserting operator randomly selects a position of the vector, and then the values after that position are replaced by new random values, as shown in Figure 5.6.

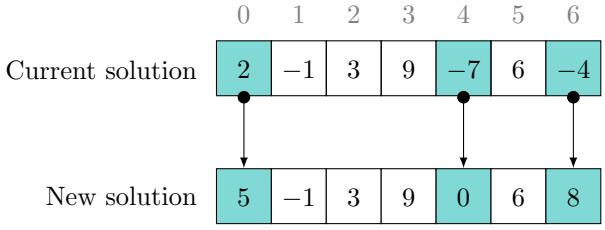


Figure 5.5: Perturbation tweaking

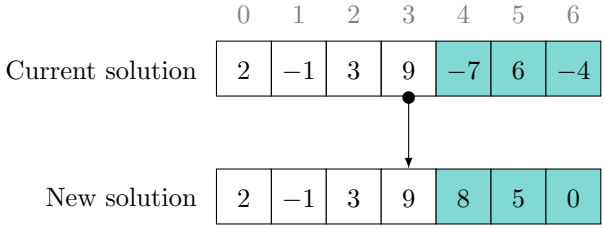


Figure 5.6: Perturbation inserting

Local Search Local search is a heuristic that attempts to improve the objective function value of the solution to which it is applied [126]. A new solution will not have a worse objective function value, but it may be the same as the original solution's value. The behaviour of these heuristics is controlled with the *depth of search* parameter, which is used to set a number of iterations for the local search [126].

Stochastic Hill Climbing In each iteration of HC, a neighbour is generated, and it is accepted immediately if it has the superior or equal fitness to the current solution; otherwise, the change is not accepted. The search is terminated when the *depth of search* is reached.

Tabu Search By maintaining a *tabu list*, TS keeps around a history of recently considered candidate solutions and refuses those candidate solutions until they are sufficiently far in the past.

The characters, which represent these defined heuristics, are listed in Table 5.1.

Table 5.1: List of low-level heuristic representatives

Character	Heuristic
a	Mutation converting
b	Mutation interchanging
c	Mutation reversing
d	Perturbation tweaking
e	Perturbation inserting
f	HC
g	TS

5.2.2 EvoHyp Parameters

Apart from the low-level heuristics that a user has to define, EvoHyp provides GA features via the parameters, including population size, tournament size, number of generations, mutation rate, crossover rate, initial maximum length, offspring maximum length and mutation length. Some essential operations associated with the parameters are further described below:

Tournament Selection Tournament selection is used to select parents which the mutation and crossover operators will then be applied to produce the offspring for the next generation [128].

Mutation Operator Mutation operator randomly selects a mutation point, and the character at that position in the chromosome is replaced with a randomly created sub-string [128]. The length of the sub-string is randomly chosen to be in the range of 1 to the mutation length [128]. Regarding Figure 5.1, for example, at a mutation point of 2 in chromosome 2, heuristic a is replaced by a newly random sub-string $\{d, e, e\}$ as presented in Figure 5.7.

Crossover Operator The crossover operator randomly selects two points in each of the parents, and the parent chromosomes are crossed over at these points to produce two offspring; however, only the fitter one of the offspring is returned [128]. The maximum length of the offspring can be specified by the offspring maximum length [128]. Regarding Figure 5.1, for example, chromosomes 2 and n are crossed over to produce two offspring as depicted in Figure 5.8.

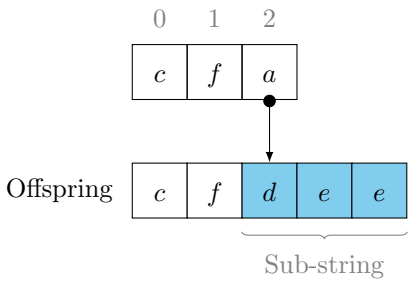


Figure 5.7: EvoHyp's mutation operator

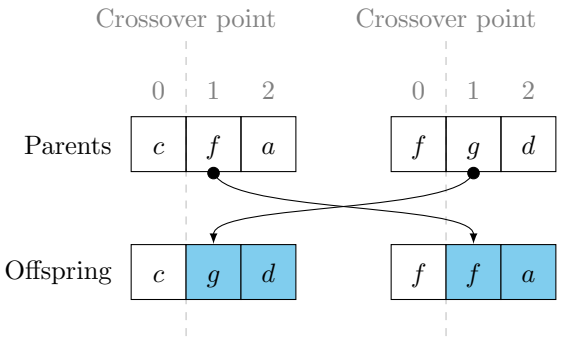


Figure 5.8: EvoHyp's crossover operator

The maximum length of offspring can be specified by parameter `offspringMaxLength`. EvoHyp also allows the offspring produced by generic operators to be no limit on its size by specifying such parameter value to -1 [128].

Furthermore, the population of each generation is created by applying the mutation rate and crossover rate [128]. For example, if the mutation rate is 0.5 and the crossover rate is 0.3, then the proportions of individuals within a population will be $50 : 30 : 20$, where the remaining 20 percentage of the population will be created using reproduction [128]. The search will be terminated when the number of generations is reached [128].

5.3 Preliminary Analysis

The preliminary analysis in this chapter is different from those former two chapters. Particularly, since other four development boards were running the experiments of Chapter 4, we started our first attempt to carry out the investigation on the remaining board by initially specifying the parameters as listed in Table 5.2 for the quartic equation in July 2017.

In particular, the values of population size, tournament size and the number of generations were given similar to those of GA in Chapter 3. Furthermore, unlike GA in general, where the new individual is produced by crossing over the parents and then mutating offspring, the crossover and mutation operations of EvoHyp independently generates the new offspring of the population into a proportion of mutation, crossover and reproduction, respectively, as earlier described in Section 5.2.2. Accordingly, we first gave the proportion of evolutionary operations to $50 : 30 : 20$. Besides, in the initial generation, the maximum length of an initial individual was given to 20. In addition, the number of iterations for local searches was initially set to 200 and the tabu list in particular was given at ten per cent of such number of iterations.

Unfortunately, about a month later, the COMX-P4080 board unexpectedly stopped working due to overheating. This overheating problem developed owing to the very long duration in which the boards were persistently stressed by the experiments. This issue occasionally occurred during the conduct of the experiments in the other chapters as well. We have maintained the development boards by shutting them down for a few days before restarting the experiment again.

Table 5.2: Parameter setting for the first and second attempts of preliminary analysis

Parameter	**Value**
`populationSize`	100
`tournamentSize`	2
`noOfGenerations`	101
`mutationRate`	0.5
`crossoverRate`	0.3
`initialMaxLength`	20
`offspringMaxLength`	-1
`mutationLength`	3
`searchDepth`	200
`tabuList`	0.1

As of the submission date of the thesis (30 September 2017), the second attempt had not yet completed. The parameter setting was also the same as in the first attempt. This time we had got another board available. So, we conducted the investigations on both the quartic and sextic equations.

As reported by the log files, the best combination of low-level heuristics, and best test inputs and its fitness value, for each preliminary experiment as of 30 September 2017 is summarised in Table 5.3.

Based on these preliminary results, compared to the results obtained by the direct approach as summarised in Table 3.5, GA-based hyper-heuristic outperformed SHC, direct GA and RS for the case of the quartic equation, whereas it could only surpass SHC for sextic equation case. However, the hyper-heuristic was worse than the indirect approach in all the cases.

According to the results summarised in Table 5.3, we have found that this highly computational time demand happens due to the two local search heuristics, i.e. HC and TS, which require a number of iterations every single time they are randomly selected as an element of a chromosome by the high-level GA mechanism of EvoHyp.

Furthermore, such computational requirements are exacerbated by the frequent selection of local search low-level heuristics. Particularly, both local search heuristics are likely to have more chance of being selected by the high-level GA mechanism than the rest because of their higher fitness values produced through hundreds of iterations.

Unsuccessfully, the preliminary investigation of the second attempt failed for the sake of the annual network and systems maintenance of the department during the Christmas holiday. All development boards were shut down.

Table 5.3: Best values of coefficients obtained by the preliminary analysis

Arguments	**Heuristics**	**Coefficient vector**	**Fitness value**
5	g; f; f; f; d and d	19,268; 15,896; 20,620; −4,401 and −14,044	95,219
7	g; c; d; c; c; a; c; b and f	12,290; 14,619; 12,347; 30,646; 29,585; 22,268 and 13,355	161,481

5.4 Experiment VI—Genetic Algorithm Hyper-Heuristic

5.4.1 Objective

The objective in this experiment is to apply a hyper-heuristic to search for test inputs that might cause a single-threaded polynomial root-finding routine running on the COMX-P4080 board to violate performance timing requirements. Although GA was inferior to those single-solution metaheuristic techniques, i.e. HC and SA, in seeking extreme test inputs as evidenced in the previous two chapters, it would be useful to also investigate its ability as a high-level methodology to search a combination of low-level heuristics, which in the end may produce the extreme test inputs. The problem statement in this experiment is, therefore, similar to Experiment I in Chapter 3, which is:

Problem Statement: *For a polynomial equation of the form* $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} = 0$*, we seek values of the coefficients (*a_0 *to* a_n*) that maximise the execution time of the polynomial solver.*

The research question addressed in this experiment is:

Research Question: *Is EA hyper-heuristic approach effective in seeking values of the coefficients that maximise the execution time of the polynomial solver?*

5.4.2 Preparation

We previously explained in Section 5.3, the computational time issue that occurred during the preliminary analysis. As a result, we decided to reduce the computational time by rerunning the experiment in this chapter with the new parameter setting shown in Table 5.4. Similar to Chapter 4, in this chapter we conducted the experiment to find the coefficients of the quartic and sextic equations that maximise the execution time of the GSL's polynomial root-finder by using EA hyper-heuristic.

5.4.3 Method

The methodology in this chapter corresponded to other experiments throughout the thesis. In particular, the hyper-heuristic was executed with each problem input size of the polynomial function ten times. Each trial was provided with a different random seed generated from `random.org`. The initial random seeds for quartic an sextic equations are 8,675 and 8,610, respectively.

Table 5.4: Parameter setting for GA-based hyper-heuristic

Parameter	Value
`populationSize`	10
`tournamentSize`	2
`noOfGenerations`	21
`mutationRate`	0.5
`crossoverRate`	0.3
`initialMaxLength`	10
`offspringMaxLength`	5
`mutationLength`	5
`searchDepth`	100
`tabuList`	0.1

5.4.4 Results

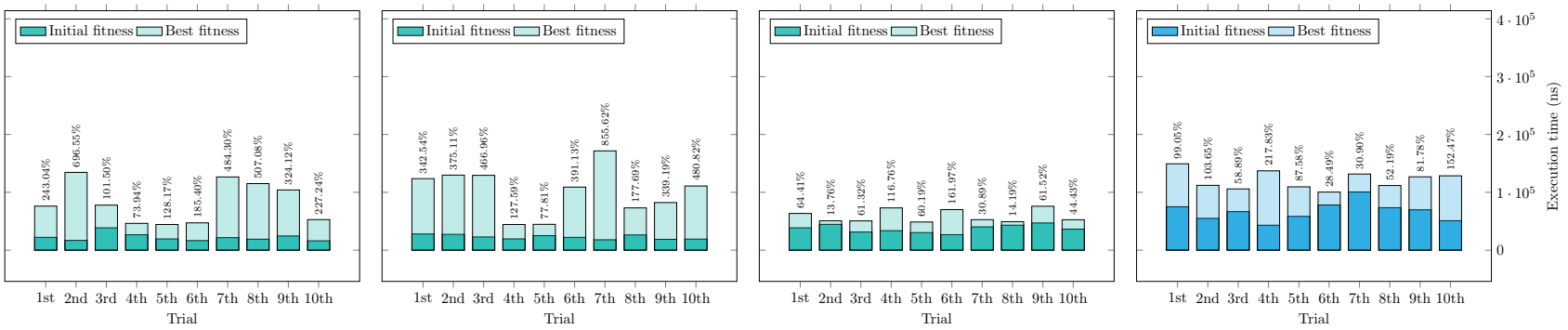
In this chapter, the results obtained by GA hyper-heuristic were compared with the results of Chapters 3 and 4. Particularly, Figures 5.9 and 5.10 display the differences between the initial and the final (best) fitness values gained from ten trials of each experiment on stressing the polynomial solver for quartic and sextic equations of direct and hyper-heuristic approaches. Since these were almost the same in terms of the characteristic of the initial fitness values among three different indirect approaches, therefore, we selected the results gained by basic two-interval based indirect to represent the differences between the initial and the final fitness values gained from ten trials of indirect and hyper-heuristic approaches as presented in Figures 5.11 and 5.12. For ease of reference, we abbreviate 'hyper-heuristic' to 'HH'.

Distributions of the best fitness values obtained from ten trials of hyper-heuristic in comparison with direct approaches are shown in Figure 5.13 and with indirect approaches are presented in Figures 5.14 and 5.15.

Finally, Table 5.5 summarises the longest execution time obtained from all search-based approaches, including direct, indirect and hyper-heuristic approaches.

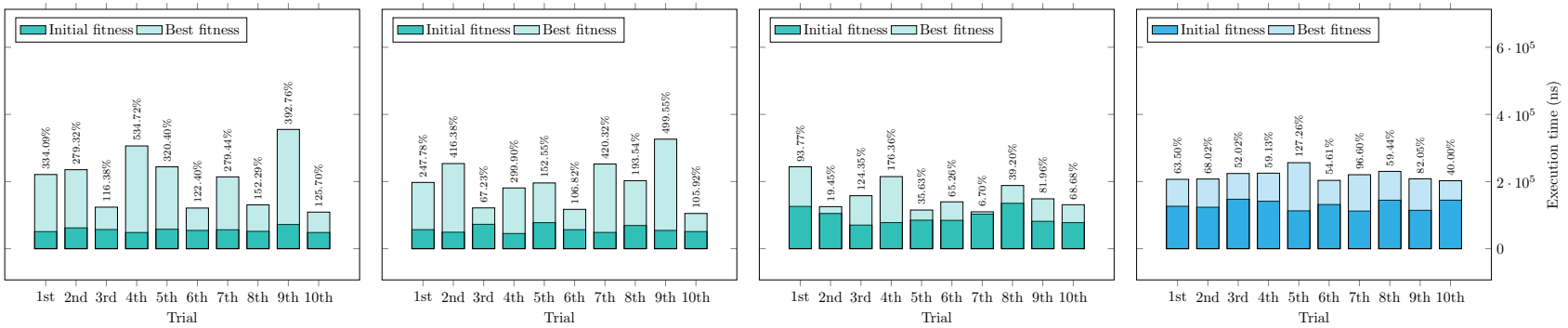
5.4.5 Discussion and Conclusions

The research question in this chapter was answered through a comparison of the effectiveness of seeking the values of coefficients that maximise the execution time of polynomial root-finder among the results attained by hyper-heuristic, direct and indirect approaches.



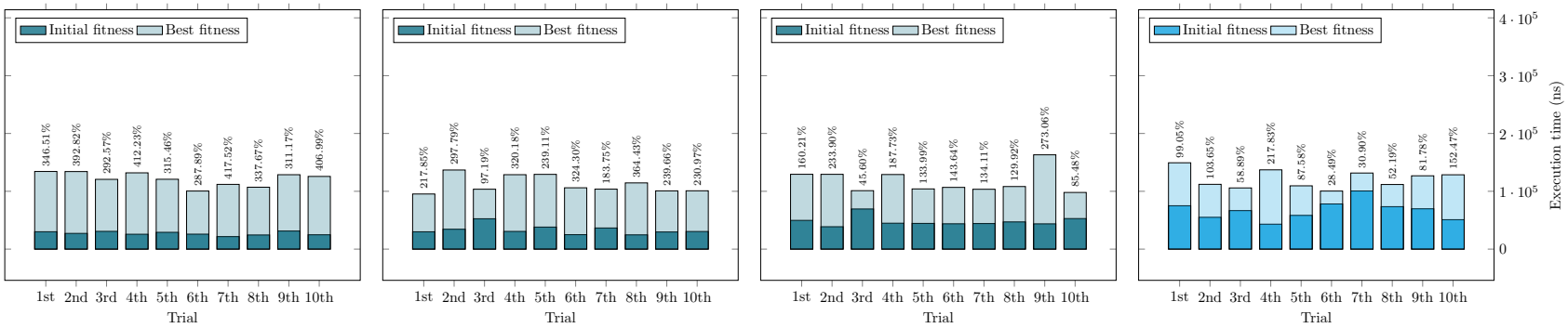
(a) Direct HC **(b)** Direct SA **(c)** Direct GA **(d)** HH

Figure 5.9: Results of 10 trails obtained by the GA-based hyper-heuristic to the quartic equation in comparison with the direct approaches



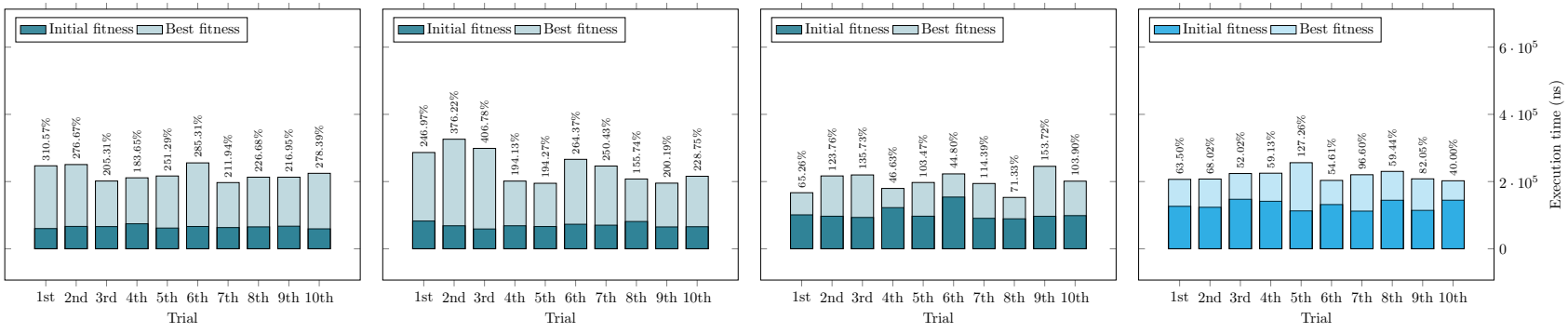
(a) Direct HC **(b)** Direct SA **(c)** Direct GA **(d)** HH

Figure 5.10: Results of 10 trails obtained by the GA-based hyper-heuristic to the sextic equation in comparison with the direct approaches



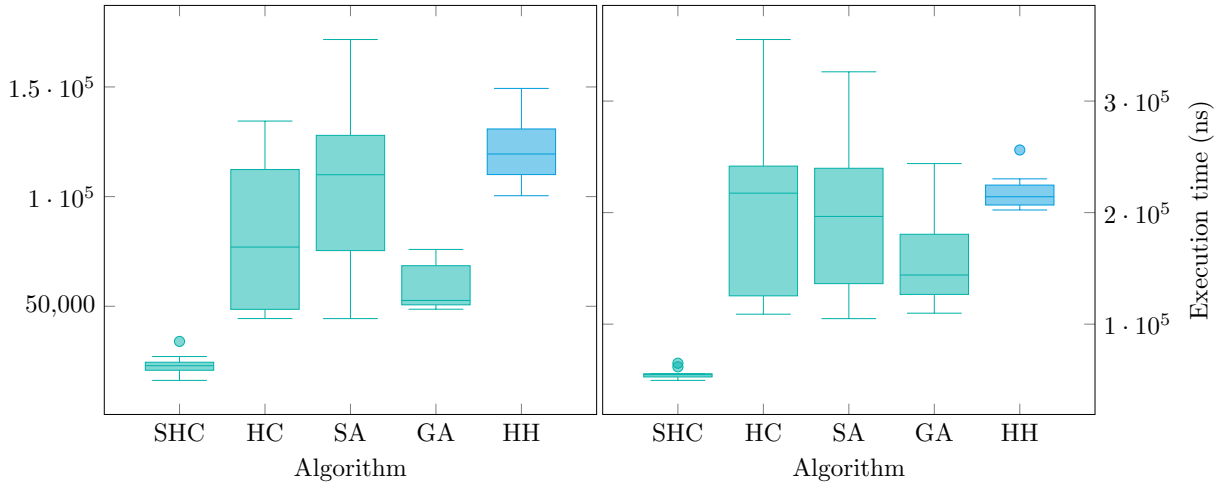
(a) Indirect(2) HC **(b)** Indirect(2) SA **(c)** Indirect(2) GA **(d)** HH

Figure 5.11: Results of 10 trails obtained by the GA-based hyper-heuristic to the quartic equation in comparison with the basic 2-interval based indirect approach



(a) Indirect(2) HC **(b)** Indirect(2) SA **(c)** Indirect(2) GA **(d)** HH

Figure 5.12: Results of 10 trails obtained by the GA-based hyper-heuristic to the sextic equation in comparison with the basic 2-interval based indirect approach



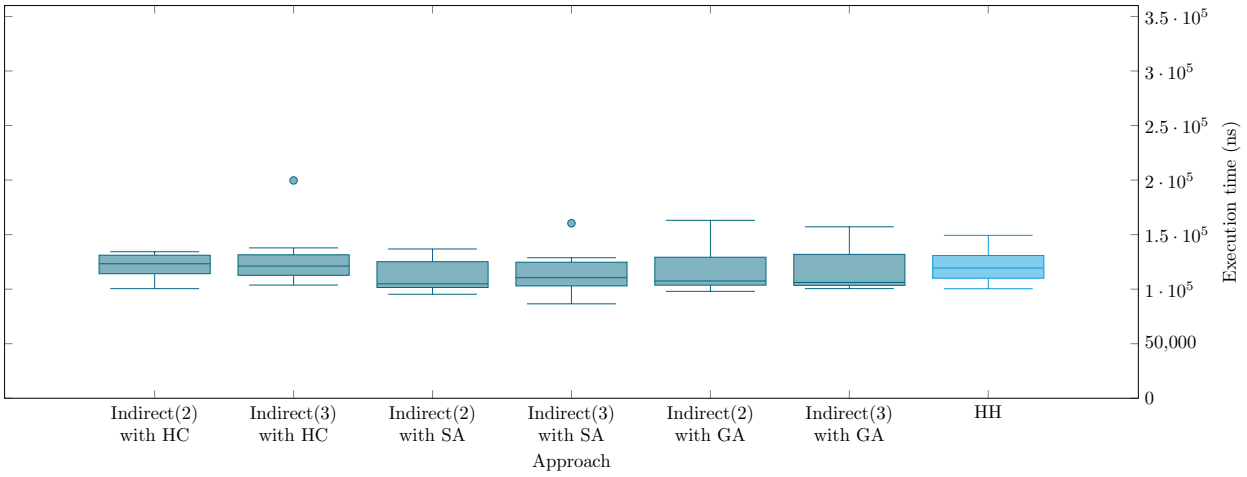
(a) Quartic equation **(b)** Sextic equation

Figure 5.13: Distribution of the best fitness values of 10 trials of the direct approaches and GA-based hyper-heuristic on the quartic and sextic equations

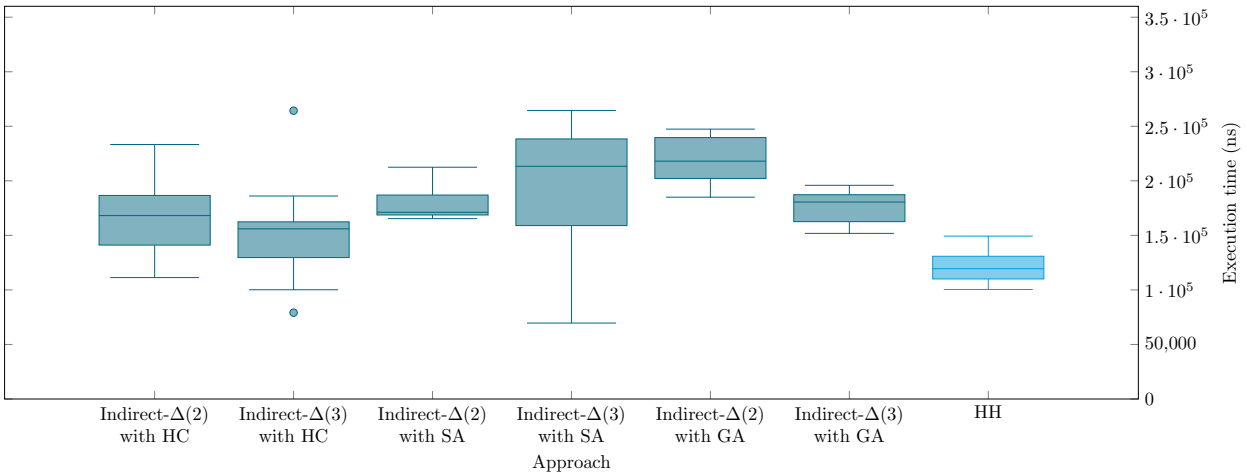
According to Figures 5.9 to 5.12, compared with direct and indirect approaches, GA hyper-heuristic started the search with higher fitness values in general. In particular, hyper-heuristic takes advantage of GA to choose the best fitness value among the initial chromosomes of its population. Even though direct GA was the second-best approach delivered high fitness values in the initial generation, its initial fitness values are gained by the initial individuals, i.e. test vectors, which are directly generated at random. On the other hand, the test vectors in the initial generation of GA hyper-heuristic have more chance of improving their fitness values since they are evolved along the series of heuristics.

Considering the final fitness values among ten trials of each approach, in comparison with direct approaches, hyper-heuristic outperformed SHC and GA, as well as had smaller variations than others (except SHC), in both cases of quartic and sextic equations as shown in Figure 5.13.

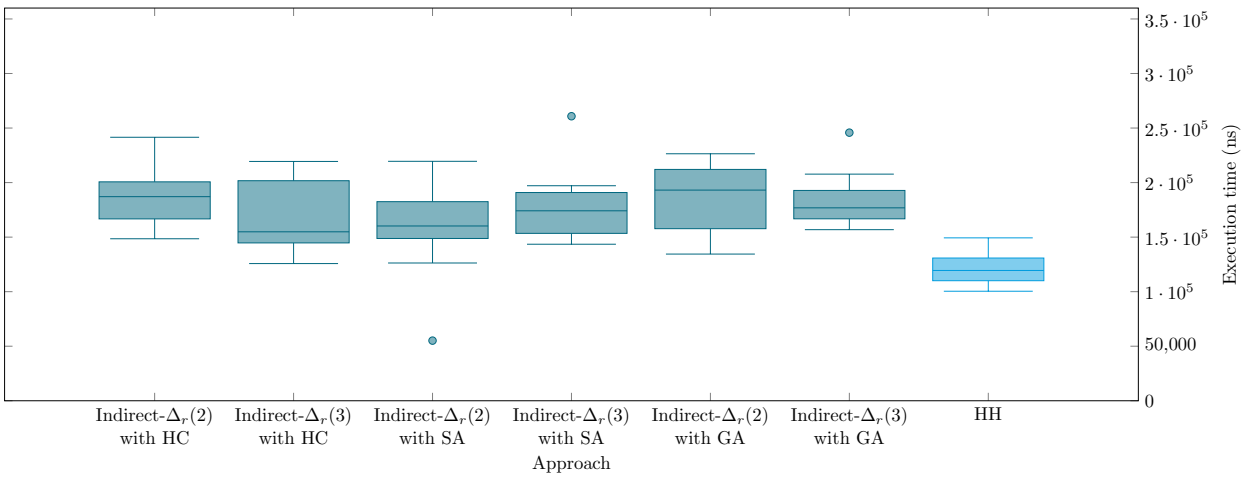
Moreover, hyper-heuristic performed roughly similar to the basic method of indirect approaches, where the intervals are variant-free, as indicated by the IQRs which are almost at the same level on the plots (some higher extreme values appeared on some indirect approaches though), as presented in Figures 5.14a and 5.15a. Only the basic two-interval based indirect approach with SA was explicitly surpassed hyper-heuristic in the case of the sextic equation.



(a) Basic interval indirect approaches

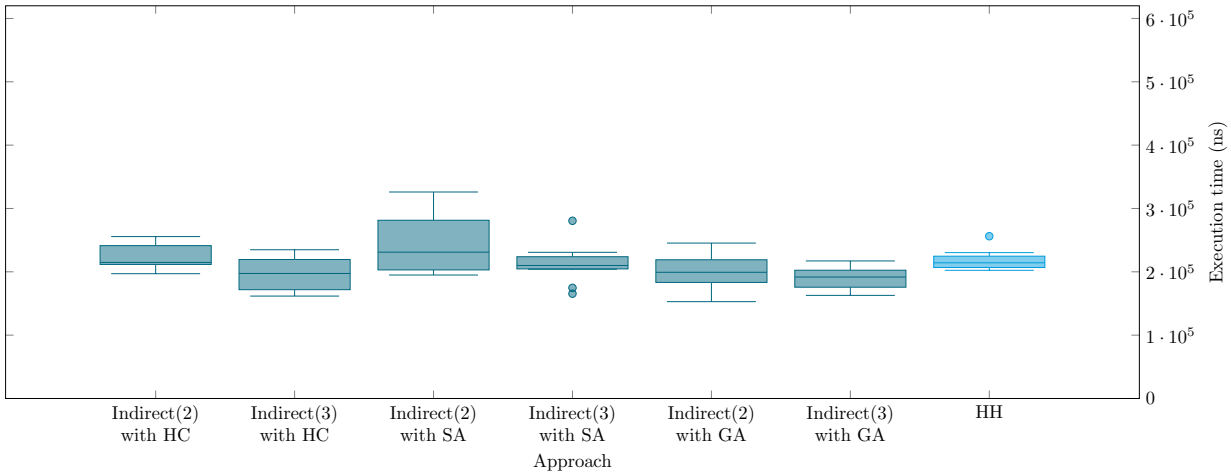


(b) Fixed delta-based interval indirect approaches

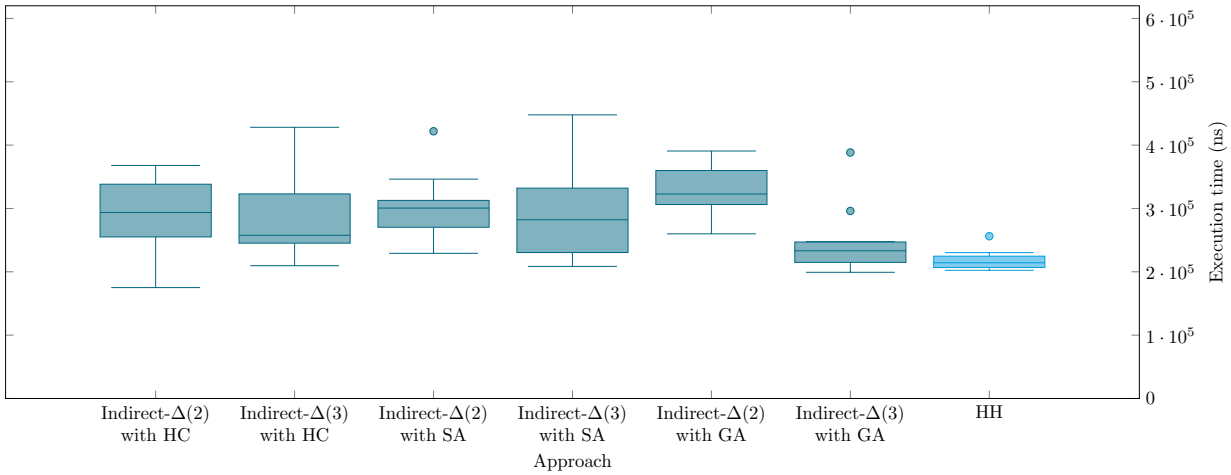


(c) Randomised delta-based interval indirect approaches

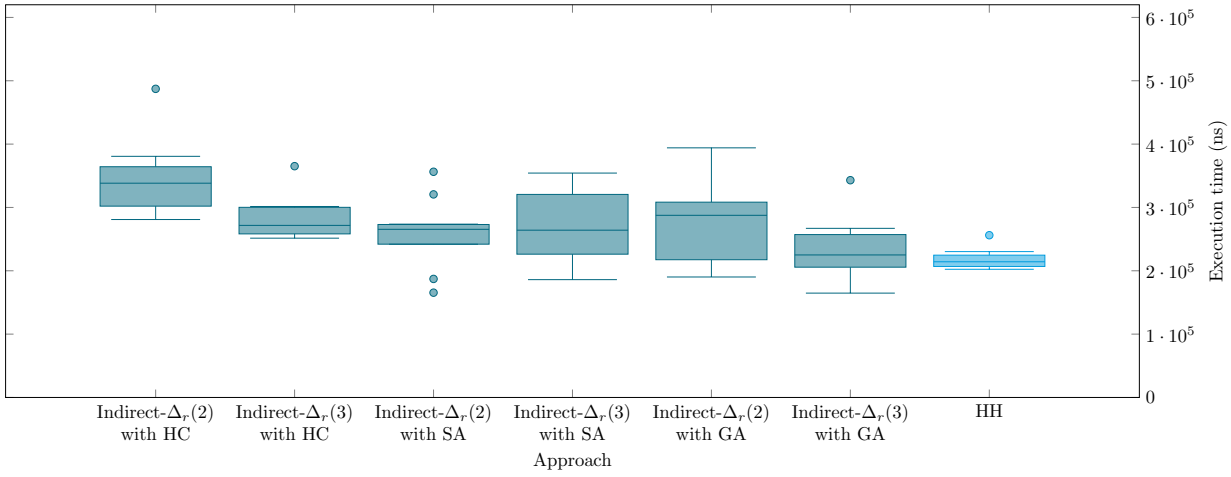
Figure 5.14: Distribution of the best fitness values of 10 trials of the indirect approaches and GA-based hyper-heuristic on the quartic equation



(a) Basic interval indirect approaches



(b) Fixed delta-based interval indirect approaches



(c) Randomised delta-based interval indirect approaches

Figure 5.15: Distribution of the best fitness values of 10 trials of the indirect approaches and GA-based hyper-heuristic on the sextic equation

Table 5.5: Summary of the highest fitness value of 10 trials of the direct, indirect and hyper-heuristic over different input arguments for the polynomial solver

Arguments	RS	SHC	HC	SA	GA	HH
5	66,978	33,980	134,449*	171,629.5*	75,917*	149,329
			264,310‡	264,523‡	247,400†	
7	204,990	64,984	355,198*	326,249*	243,980*	256,156
			487,240§	447,760‡	394,097§	

* Direct. † Indirect-$\Delta(2)$. ‡ Indirect-$\Delta(3)$. § Indirect-$\Delta_r(2)$.

However, as depicted in Figures 5.14b, 5.14c, 5.15b and 5.15c, when comparing with the more restricted versions of indirect approaches, i.e. fixed delta-based and randomised delta-based intervals, hyper-heuristic was the worst in all cases.

Overall, hyper-heuristic was mostly superior to direct approaches but worse than indirect approaches in seeking temporal coefficient vectors for polynomial solver on quartic and sextic equations as summarised in Table 5.5. In terms of computational demands, however, GA hyper-heuristic is considered inefficient. Particularly, based on the termination criterion, i.e. the number of generations, both direct and indirect approaches are remarkably straightforward. No matter how many experimental trials are conducted, each trial takes approximately the same computational time because an individual in general, or a test vector in particular, is directly generated either by a metaheuristic for the direct approach or by a selected sampling regime for the indirect approach.

For GA hyper-heuristic, on the other hand, its test vector is a result of a chromosome, which is a combination of the selected heuristics. The computational time for each experimental trial thus depends on how many elements are there in each chromosome, as well as what have selected heuristics in the chromosome. For example, in this experiment, local searches take relatively 100 times longer than other low-level heuristics due to their number of iterations. Table 5.6 summarises the computational time used in each trial of GA hyper-heuristic. Note: the time taken is presented in the format $hours : minutes : seconds$.

In this experiment, the best coefficient vectors among ten trials for quartic and sextic equations, together with the best fitness values and the best chromosomes, are summarised in Table 5.7. Additionally, the numbers of QR iterations, which maximised the execution times of GSL's root-finder, are also included in the table.

Table 5.6: Computational demands of 10 trials of GA hyper-heuristic

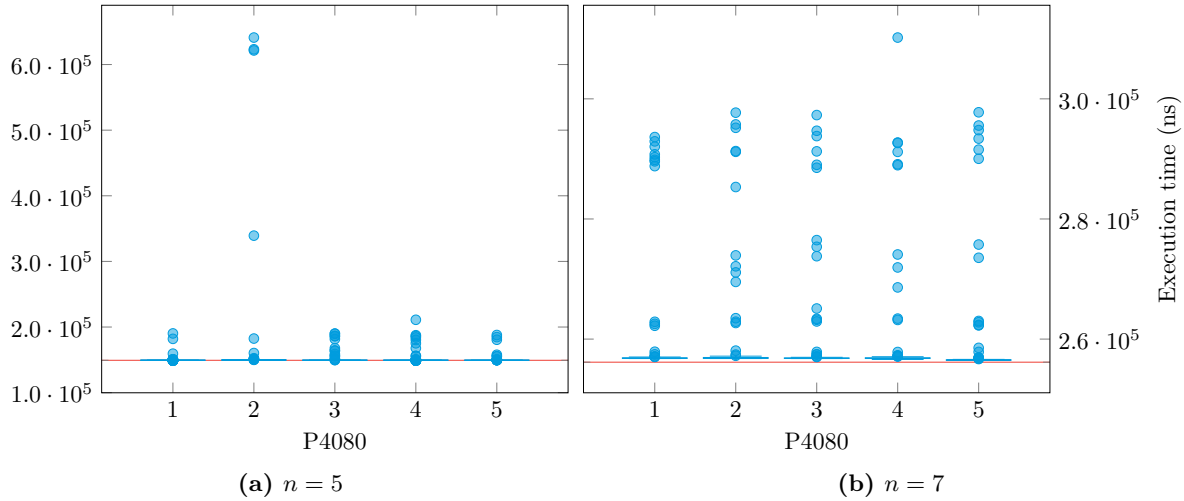
	Arguments	
Trial	**5**	**7**
1	45:27:10	67:08:53
2	49:28:58	62:04:17
3	48:40:26	75:01:30
4	53:27:04	70:36:44
5	39:55:18	49:05:49
6	57:43:32	62:00:00
7	57:37:07	80:52:50
8	78:32:49	75:27:51
9	50:25:49	64:34:46
10	47:44:24	56:17:33

Table 5.7: Best values of coefficients (Chapter 5)

Arguments	Heuristics	Coefficient vector	Iterations
5	g; f and f	−22,783; 3,042; 31,270; −4,245 and −16,092	52
7	f; e; f; f and f	28,689; 17,413; 1,102; −9,699; −22,566; −1,507 and 13,888	48

Regarding the best combinations of low-level heuristics presented in Table 5.7, HC was largely selected as parts of such best chromosomes. Owing to the fact that the new solution gained by local searches will not have worse objective function value as previously described in Section 5.2.1, hence, HC and TS are likely to be picked up more often than other low-level heuristics, but they can lead to a longer computing time at the same time.

Likewise Chapters 3 and 4, we validated these best coefficient vectors by rerunning them on five different P4080 boards. Figure 5.16 show box plots of the execution times collected from 100 runs of each coefficient vector on each board. As indicated by the midspreads on the box plots, there were almost no variations over the distributions among five boards. Those IQRs were exactly at the same level in the case of the quartic equation and were almost at the same level in the case of the sextic equation. The actual execution times are also within such midspreads.



(a) $n = 5$ **(b)** $n = 7$

Figure 5.16: Distribution of 100 execution times of polynomial solver with best input arguments obtained from 5 P4080 boards (Chapter 5)

5.5 Summary

In conclusion, this chapter explored the ability of a selective perturbative hyper-heuristic to seek temporal input arguments that maximise the execution time of a task running on a multicore system. In particular, a GA is used as a high-level methodology to select a combination of low-level perturbative heuristics, which generates extremal test inputs for the polynomial solver. Accordingly, we defined seven low-level heuristics, which are converting mutator, interchanging mutator, reversing mutator, tweaking operator, inserting operator, HC and TS.

Regarding Table 5.5, compared to the direct approaches, the experimental results indicate that test inputs (or a coefficient vector) produced by a combination of low-level heuristics, which are selected by an EA hyper-heuristic, delivered a higher fitness value than the fitness value of test inputs, which are directly generated from RS, SHC, HC and GA for finding roots of the quartic equation. However, in the case of finding seven coefficients of the sextic function, the fitness value produced by the test inputs generated from a combination of selected heuristics of the EA hyper-heuristic was only greater than the value from test inputs, which were directly generated by RS, SHC and GA. Besides, such GA hyper-heuristic was the worst in comparison with the indirect approaches when the intervals of indirect approaches are restricted by delta values.

In terms of computational time, using a population-based approach as a high-level methodology of a hyper-heuristic would appear to be significantly inefficient for our timing-related problem. Since every time a set of heuristics are repeatedly selected and combined with other heuristics (or set of heuristics), the GA mechanism requires a re-evaluation of such a set of heuristics again as it is considered to be a new chromosome of the population, especially when those two local search heuristics are picked up. This leads to a huge demand for computation time required to perform such GA hyper-heuristic. A single-solution approach probably may be more suitable for the timing-related problem than the population-based approach. As evidenced throughout the thesis, the single-solution based approaches are remarkable in attaining extreme execution times for temporal testing, either as the direct approaches to generate the evolved test data or as the indirect approaches to generate the evolved sampling distributions.

In the research of this chapter, more than that of the previous chapters, the computational demands have proven problematic. Exploring the attainment of extreme execution times on the P4080 platform is well motivated in general, but perhaps the evaluation of hyper-heuristics is computationally just a step too far. It would be possible to research hyper-heuristics more confidently if a cloud-based solution was adopted, e.g. if we were run on a set of identical specific multicore chips provided by, say, Amazon Web Services (AWS). (Note: Amazon and other cloud computing providers do not offer P4080 provision.)

Chapter 6

Conclusions

This thesis has explored the use of various forms of optimisation approaches to the generation of test inputs exhibiting extreme execution times on a modern multicore platform. The work has addressed:

1. The direct maximisation of the execution time.

2. The indirect maximisation of execution time, via the synthesis of correlated sampling strategies.

3. The hyper-heuristic maximisation of extreme execution time (WCET).

6.1 Contributions

The contributions of the work described here are:

1. The most wide-ranging comparison of optimisation techniques to finding extremal execution times we know of (and executing on a multicore chip).

2. The first investigation of optimisation approaches to the generation of dependent input sampling strategies for test input sampling in the context of extreme execution times. (Our particular target was a specific multicore chip, the P4080, but we know of no other such application for any chip.)

3. The first investigation of the use of hyper-heuristics to generate extreme execution times. (Our particular target was a specific multicore chip, but we know of no other such application for any chip.)

Furthermore, the work has demonstrated that the indirect approach formulated and explored has considerable merit.

6.2 Limitations of the Research

The limitations of the work are:

1. The small number of case studies we were able to use. More confident general results might be inferred from a much larger corpus of examples.

2. The availability of computational resources for our specific chip of choice has distinctly limited the amount of computation we were able to do. The choice of the specific chip (i.e. P4080) was well motivated; being in active use in critical platforms. However, the choice of this chip largely means that we are limited to the actual number of chips procured for the research. Our evaluation environment started with a single specialised P4080 board, but it became clear that this simply would not suffice. We finally had five specially constructed P4080 systems for cost function evaluation at the time of completion of this thesis. Because we wish to comment on the usefulness of an approach we need to do meaningful numbers of experiments, which itself boosts the required computation resources hugely. Using five boards is a substantial computational resource, but still, the computational demands of the research are very significant. In the final nine months of the thesis, constant running of all five boards was the norm. Computational limitations were felt particularly for the work on Chapter 5 (hyper-heuristics), but the issue is relevant to all the work reported this thesis. Over the empirical evaluation of the research, I estimate that 14,393 P4080 computational hours have been expended. The estimation excludes time expended in failed experiments due to overheating, as well as in the preliminary analysis, especially the ones in Chapter 5, which took almost a half a year with failures.

3. More complex data types need to be considered. Our case study problems are numerical with integral inputs. The issue of complex data types is an acknowledged issue in SBST.

4. The lack of a Java Virtual Machine (JVM) for the COMX-P4080 board was a limitation. This forced the separation of computation for the optimisation technique

(carried out on a desktop PC) from the cost function evaluation (which was done on P4080 machines). Naturally, this incurs communications overheads.

5. The very limited evaluation of hyper-heuristics is a distinct although easily fixable limitation. We simply ran out of time for further carrying out experiments with various parameter settings.

6.3 Future Work

The future work in this area should include:

1. Distribution over the cloud to effect large-scale experimenting. This would enable truly rigorous experimentation, albeit for a multicore environment used commercially rather than in critical systems. For example, we could farm out evaluations to AWS or similar.

2. Simulation and virtualisation is one possible solution, effectively adopting a P4080 simulator as the execution platform and distributing many instances across the cloud. However, it is far from clear what constraints this approach would impose. Simulators are only a model of the system and in terms of execution time (of the simulator) may well be terribly slow. However, this option should at least be considered for future work.

3. Wider use of multi-threaded applications in the case studies.

4. Wider exploitation of the techniques researched to generate test data to induce other extreme performance measures. There seems little reason to believe a technique that generated extreme execution times would not find reasonable application to, for example, generation of extreme power usage.

5. Wider exploitation of the indirect approach by utilising other probability distributions, such as beta, binomial, normal, exponential and Weibull distributions, to generate test data from selected subdomains. The parameters of distributions for sampling subdomains could readily be incorporated in the optimisation process. (Distributional parameters could, for example, easily be incorporated in the chromosome representation indicated in Section 4.2 and decoded appropriately.)

6. Wider exploitation of the hyper-heuristic approach using a single-state solution approach, which would be an alternative to finding extreme execution times.

7. Manually crafted hybrids. Rather than using a hyper-heuristic to find a hybrid we could investigate specific combinations directly.

6.3.1 Looking Forward

The area of temporal testing appears to be a promising avenue for search-based approaches. There is clear value in further exploring the higher level search spaces of Chapters 4 and 5. The area is recommended to the research community.

Appendix A

Benchmark Source Codes

This appendix shows pieces of codes of the SUTs used throughout the thesis.

A.1 Polynomial Root Finder

The code fragments related to the polynomial root-finding routine, i.e. `zsolve.c` and `qr.c`, are displayed in Listings A.1 and A.2. The full versions of these codes are available to be downloaded at [129].

Listing A.1: `zsolve.c` [129]

```
1  #define MAT(m,i,j,n) ((m)[(i)*(n) + (j)])
2
3  #include "companion.c"
4  #include "balance.c"
5  #include "qr.c"
6
7  int
8  gsl_poly_complex_solve (const double *a, size_t n,
9                          gsl_poly_complex_workspace * w,
10                         gsl_complex_packed_ptr z)
11 {
12   int status;
13   double *m;
14
15   if (n == 0) {
16     GSL_ERROR("number of terms must be a positive integer", GSL_EINVAL);
17   }
18
```

```
19    if (n == 1) {
20      GSL_ERROR("cannot solve for only one term", GSL_EINVAL);
21    }
22
23    if (a[n - 1] == 0) {
24      GSL_ERROR("leading term of polynomial must be non-zero", GSL_EINVAL);
25    }
26
27    if (w->nc != n - 1) {
28      GSL_ERROR("size of workspace does not match polynomial", GSL_EINVAL);
29    }
30
31    m = w->matrix;
32
33    set_companion_matrix (a, n - 1, m);
34
35    balance_companion_matrix (m, n - 1);
36
37    status = qr_companion (m, n - 1, z);
38
39    if (status) {
40      GSL_ERROR("root solving qr method failed to converge", GSL_EFAILED);
41    }
42
43    return GSL_SUCCESS;
44  }
```

Listing A.2: `qr.c` [129]

```
1  static int
2  qr_companion (double *h, size_t nc, gsl_complex_packed_ptr zroot)
3  {
4    double t = 0.0;
5
6    size_t iterations, e, i, j, k, m;
7
8    double w, x, y, s, z;
9
10   double p = 0, q = 0, r = 0;
11
12   /* FIXME: if p,q,r, are not set to zero then the compiler complains
13      that they ``might be used uninitialized in this
14      function''. Looking at the code this does seem possible, so this
15      should be checked. */
16
17   int notlast;
18
19   size_t n = nc;
20
21 next_root:
22
23   if (n == 0)
24     return GSL_SUCCESS ;
25
26   iterations = 0;
27
28 next_iteration:
29
30   for (e = n; e >= 2; e--)
31     {
32       double a1 = fabs (FMAT (h, e, e - 1, nc));
33       double a2 = fabs (FMAT (h, e - 1, e - 1, nc));
34       double a3 = fabs (FMAT (h, e, e, nc));
35
36       if (a1 <= GSL_DBL_EPSILON * (a2 + a3))
37         break;
38     }
39
40   x = FMAT (h, n, n, nc);
```

```
41
42    if (e == n)
43      {
44        GSL_SET_COMPLEX_PACKED (zroot, n-1, x + t, 0); /* one real root */
45        n--;
46        goto next_root;
47        /*continue;*/
48      }
49
50    y = FMAT (h, n - 1, n - 1, nc);
51    w = FMAT (h, n - 1, n, nc) * FMAT (h, n, n - 1, nc);
52
53    if (e == n - 1)
54      {
55        p = (y - x) / 2;
56        q = p * p + w;
57        y = sqrt (fabs (q));
58
59        x += t;
60
61        if (q > 0)                  /* two real roots */
62          {
63            if (p < 0)
64              y = -y;
65            y += p;
66
67            GSL_SET_COMPLEX_PACKED (zroot, n-1, x - w / y, 0);
68            GSL_SET_COMPLEX_PACKED (zroot, n-2, x + y, 0);
69          }
70        else
71          {
72            GSL_SET_COMPLEX_PACKED (zroot, n-1, x + p, -y);
73            GSL_SET_COMPLEX_PACKED (zroot, n-2, x + p, y);
74          }
75        n -= 2;
76
77        goto next_root;
78        /*continue;*/
79      }
80
81    /* No more roots found yet, do another iteration */
```

```
82
83    if (iterations == 120)  /* increased from 30 to 120 */
84      {
85        /* too many iterations - give up! */
86
87        return GSL_FAILURE ;
88      }
89
90    if (iterations % 10 == 0 && iterations > 0)
91      {
92        /* use an exceptional shift */
93
94        t += x;
95
96        for (i = 1; i <= n; i++)
97          {
98            FMAT (h, i, i, nc) -= x;
99          }
100
101       s = fabs (FMAT (h, n, n - 1, nc)) + fabs (FMAT (h, n - 1, n - 2, nc));
102       y = 0.75 * s;
103       x = y;
104       w = -0.4375 * s * s;
105     }
106
107   iterations++;
108
109   for (m = n - 2; m >= e; m--)
110     {
111       double a1, a2, a3;
112
113       z = FMAT (h, m, m, nc);
114       r = x - z;
115       s = y - z;
116       p = FMAT (h, m, m + 1, nc) + (r * s - w) / FMAT (h, m + 1, m, nc);
117       q = FMAT (h, m + 1, m + 1, nc) - z - r - s;
118       r = FMAT (h, m + 2, m + 1, nc);
119       s = fabs (p) + fabs (q) + fabs (r);
120       p /= s;
121       q /= s;
122       r /= s;
```

```
123
124         if (m == e)
125           break;
126
127         a1 = fabs (FMAT (h, m, m - 1, nc));
128         a2 = fabs (FMAT (h, m - 1, m - 1, nc));
129         a3 = fabs (FMAT (h, m + 1, m + 1, nc));
130
131         if (a1 * (fabs(q) + fabs(r)) <= GSL_DBL_EPSILON * fabs(p) * (a2 + a3))
132           break;
133       }
134
135     for (i = m + 2; i <= n; i++)
136       {
137         FMAT (h, i, i - 2, nc) = 0;
138       }
139
140     for (i = m + 3; i <= n; i++)
141       {
142         FMAT (h, i, i - 3, nc) = 0;
143       }
144
145     /* double QR step */
146
147     for (k = m; k <= n - 1; k++)
148       {
149         notlast = (k != n - 1);
150
151         if (k != m)
152           {
153             p = FMAT (h, k, k - 1, nc);
154             q = FMAT (h, k + 1, k - 1, nc);
155             r = notlast ? FMAT (h, k + 2, k - 1, nc) : 0.0;
156
157             x = fabs (p) + fabs (q) + fabs (r);
158
159             if (x == 0)
160               continue;             /* FIXME????? */
161
162             p /= x;
163             q /= x;
```

```
164             r /= x;
165           }
166
167         s = sqrt (p * p + q * q + r * r);
168
169         if (p < 0)
170           s = -s;
171
172         if (k != m)
173           {
174             FMAT (h, k, k - 1, nc) = -s * x;
175           }
176         else if (e != m)
177           {
178             FMAT (h, k, k - 1, nc) *= -1;
179           }
180
181         p += s;
182         x = p / s;
183         y = q / s;
184         z = r / s;
185         q /= p;
186         r /= p;
187
188         /* do row modifications */
189
190         for (j = k; j <= n; j++)
191           {
192             p = FMAT (h, k, j, nc) + q * FMAT (h, k + 1, j, nc);
193
194             if (notlast)
195               {
196                 p += r * FMAT (h, k + 2, j, nc);
197                 FMAT (h, k + 2, j, nc) -= p * z;
198               }
199
200             FMAT (h, k + 1, j, nc) -= p * y;
201             FMAT (h, k, j, nc) -= p * x;
202           }
203
204         j = (k + 3 < n) ? (k + 3) : n;
```

```
205
206         /* do column modifications */
207
208         for (i = e; i <= j; i++)
209           {
210             p = x * FMAT (h, i, k, nc) + y * FMAT (h, i, k + 1, nc);
211
212             if (notlast)
213               {
214                 p += z * FMAT (h, i, k + 2, nc);
215                 FMAT (h, i, k + 2, nc) -= p * r;
216               }
217             FMAT (h, i, k + 1, nc) -= p * q;
218             FMAT (h, i, k, nc) -= p;
219           }
220       }
221
222     goto next_iteration;
223 }
```

A.2 Sortings

Code snippets of parallel sorting routines are shown below:

A.2.1 Bubble Sort

Listing A.3 presents a code fragment of parallel bubble sort. The full version of the code is available to be downloaded at [110].

Listing A.3: `bubblesort.c` [110]

```
1  typedef struct{
2   int id;
3   int* swapCount;
4   pthread_mutex_t* swapLock;
5   pthread_barrier_t* phaseBarrier;
6   Set theSet;
7   int size;
8   int numThreads;
9   int (*compare)(Element,Element);
10 } ptEO_Data;
11
12 void pthreadsBubbleSort( Set set, int size,
13         int (*compare)(Element,Element), int numThreads )
14 {
15  int i;
16  int swapCount = 0;
17
18  // initialize the mutex
19  pthread_mutex_t swapLock;
20  pthread_mutex_init( &swapLock, 0 );
21
22  // initialize the barrier
23  pthread_barrier_t phaseBarrier;
24  pthread_barrier_init(&phaseBarrier, 0, numThreads);
25
26  // create and initialize the threads.
27  pthread_t* threads = malloc( numThreads*sizeof( pthread_t ) );
28
29  // this info is constant between threads.
30  ptEO_Data tmp;
31  tmp.swapCount = &swapCount;
```

```
32   tmp.swapLock = &swapLock;
33   tmp.phaseBarrier = &phaseBarrier;
34   tmp.theSet = set;
35   tmp.size = size;
36   tmp.numThreads = numThreads;
37   tmp.compare = compare;
38
39   for(i = 0; i < numThreads; ++i)
40   {
41
42    // assign id;
43    tmp.id = i;
44
45    // copy the tmp into its own data structure.
46    ptEO_Data* data = malloc( sizeof( ptBS_Data ) );
47    memcpy( data, &tmp, sizeof( tmp ) );
48
49    // create the thread;
50    pthread_create( &threads[i], 0, &bubbleSortCallBack, data);
51   }
52
53   // join the threads.
54   for(i = 0; i < numThreads; ++i )
55   {
56    pthread_join( threads[i], 0 );
57   }
58
59   return;
60  }
61
62  void* bubbleSortCallBack( void* in )
63  {
64   // cast our void* input to the correct type.
65   ptEO_Data* data = (ptEO_Data*) in;
66
67   // extract some data from the structure for readability.
68   Set set = data->theSet;
69   int* swapCount = data->swapCount;
70   int id = data->id;
71   int size = data->size;
72   int (*compare)(Element,Element) = data->compare;
```

```
73  int numThreads = data->numThreads;
74  int elPerThread = data->size/numThreads;
75
76  // some thread-local variables we will need.
77
78  int i; // for use in loops.
79  int numSwaps; // the number of swaps we performed in the phase.
80  Element tmp; // a temporary element for use in swaps.
81
82  int start; // starting index for this thread's chunk of the set
83  int stop; // non-inclusive stopping index
84
85  // calculate our starting and stopping indices
86  start = id*(elPerThread);
87  stop = start + elPerThread;
88
89  if( (numThreads - 1) == id)
90  {
91   stop += (size - numThreads*elPerThread);
92  }
93
94  // do the sorting
95  do
96  {
97   numSwaps = 0;
98
99   // --- even phase
100  // do compare swap operations for each even indexed element in our chunk
101  // with its left neighbor in the set.
102  for(i = start; i < stop; ++i )
103  {
104   if( i > 0
105    &&  0 == (i % 2) )
106   {
107    if( 1 == compare( set[i-1], set[i] ) )
108    {
109     tmp = set[i-1];
110     set[i-1] = set[i];
111     set[i] = tmp;
112
113     ++numSwaps;
```

```
114       }
115      }
116     }
117
118     // barrier to wait until all threads have finished even stage.
119     pthread_barrier_wait( data->phaseBarrier );
120
121     printf("id = %d, past barrier \n", id);
122
123     // --- odd phase
124     // do compare swap operations for each odd indexed element in our chunk
125     // with its left neighbor in the set.
126     for(i = start; i < stop ; ++i )
127     {
128      if( i < size
129       && 1 == (i % 2) )
130      {
131       if( 1 == compare( set[i-1], set[i] ) )
132       {
133        tmp = set[i-1];
134        set[i-1] = set[i];
135        set[i] = tmp;
136
137        ++numSwaps;
138       }
139      }
140     }
141
142     // one thread resets swapCount
143     if( 0 == id )
144     {
145      *swapCount = 0;
146     }
147
148     //  barrier to wait until all threads have finished odd stage.
149     pthread_barrier_wait( data->phaseBarrier );
150
151     // --- swap count
152     if( numSwaps > 0)
153     {
154      pthread_mutex_lock( data->swapLock );
```

```
155     {
156       *swapCount += numSwaps;
157     }
158     pthread_mutex_unlock( data->swapLock );
159   }
160
161   // barrier to wait until all threads have finished count stage.
162   pthread_barrier_wait( data->phaseBarrier );
163
164  } while(*swapCount > 0 );
165 }
```

A.2.2 Shell Sort

In Listing A.4, a code fragment of parallel shell sort is illustrated. The full version of the code is available to be downloaded at [111].

Listing A.4: `shellPT.c` [111]

```
1  void *shellsort(void *arg)
2  {
3    thread_data_t *threadWork=(thread_data_t*) arg;
4    int *a;
5    int N;
6
7    int group;
8    int i, j, h;
9    int v;
10
11   a = threadWork->a;
12   N = threadWork->N;
13   pthread_barrier_wait(&barrier);
14
15   for (h = 1; h <= N/9; h = 3*h+1) ;
16   for (; h > 0; h /= 3)
17   {
18     for (group = threadWork->threadId; group < h; group += numThreads)
19       for (i = group+h; i < N; i += h)
20       {
21         v = a[i];
22         j = i;
23         while (j >= h && a[j-h] > v)
24         {
25           a[j] = a[j-h];
26           j -= h;
27         }
28         a[j] = v;
29       }
30     pthread_barrier_wait(&barrier);
31   }
32   pthread_barrier_wait(&barrier);
33
34   return (void *) threadWork;
35 }
```

A.2.3 Quicksort

Listing A.5 depicts a code fragment of parallel quicksort. The full version of the code is available to be downloaded at [111].

Listing A.5: `qsortPT.c` [111]

```
1  int *arr;
2  int arrSize;
3  int zeroActive = 1, partitionAvailable = 0, leftEnd, rightEnd;
4  int maxThreshold;
5
6  int partition(arr, p, r)
7  int *arr, p, r;
8  {
9    int x, i, j, temp;
10
11   x = arr[p];
12   i = p - 1;
13   j = r + 1;
14   while (1)
15   {
16     while (arr[--j] > x);
17     while (arr[++i] < x);
18     if (i < j)
19     {
20       temp = arr[i];
21       arr[i] = arr[j];
22       arr[j] = temp;
23     }
24     else
25     {
26       return j;
27     }
28   }
29 }
30
31 void quickSort(int *a, int p, int r)
32 {
33   int q;
34
35   if (p < r)
```

```
36    {
37      q = partition(a, p, r);
38      quickSort(a, p, q);
39      quickSort(a, q+1, r);
40    }
41  }
42
43  void quickSort2(int *a, int p, int r)
44  {
45    int q, k;
46
47    if (p < r)
48    {
49      q = partition(a, p, r);
50      k = r - p + 1;
51      if (partitionAvailable || k > maxThreshold)
52        quickSort2(a, p, q);
53      else
54      {
55        leftEnd = p;
56        rightEnd = q;
57        partitionAvailable = 1;
58      }
59
60      k = r - q;
61      if (partitionAvailable || k > maxThreshold)
62        quickSort2(a, q+1, r);
63      else
64      {
65        leftEnd = q+1;
66        rightEnd = r;
67        partitionAvailable = 1;
68      }
69    }
70  }
71
72  void *quickSort0(void *arg)
73  {
74    thread_data_t *threadWork = (thread_data_t*)arg;
75
76    pthread_barrier_wait(&barrier);
```

```
77
78    quickSort2(arr, 0, arrSize-1);
79    zeroActive = 0;
80
81    pthread_barrier_wait(&barrier);
82
83    return (void *) threadWork;
84  }
85
86  void *quickSort1(void *arg)
87  {
88    thread_data_t *threadWork = (thread_data_t*)arg;
89
90    pthread_barrier_wait(&barrier);
91
92    while (zeroActive || partitionAvailable)
93      if (partitionAvailable)
94      {
95        quickSort(arr, leftEnd, rightEnd);
96        partitionAvailable = 0;
97      }
98
99    pthread_barrier_wait(&barrier);
100
101   return (void *) threadWork;
102 }
```

A.2.4 Merge Sort

A code fragment of parallel merge sort is demonstrated in Listing A.6. The full version of the code is available to be downloaded at [112].

Listing A.6: `parallelMergesort.c` [112]

```
1  // parallel mergesort top level:
2  // instantiate parallelMergesortHelper thread, and that's basically it.
3
4  void parallelMergesort(int lyst[], int size, int tlevel)
5  {
6   int rc;
7   void *status;
8
9   int *back = (int *) malloc(size*sizeof(int));
10
11  //Want joinable threads (usually default).
12  pthread_attr_t attr;
13  pthread_attr_init(&attr);
14  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
15
16  //pthread function can take only one argument, so struct.
17  struct thread_data td;
18  td.lyst = lyst;
19  td.back = back;
20  td.low = 0;
21  td.high = size - 1;
22  td.level = tlevel;
23
24  //The top-level thread.
25  pthread_t theThread;
26  rc = pthread_create(&theThread, &attr, parallelMergesortHelper,
27       (void *) &td);
28  if (rc)
29  {
30       printf("ERROR; return code from pthread_create() is %d\n", rc);
31       exit(-1);
32      }
33
34  //Now join the thread (wait, as joining blocks) and quit.
35  pthread_attr_destroy(&attr);
```

```
36   rc = pthread_join(theThread, &status);
37   if (rc)
38   {
39     printf("ERROR; return code from pthread_join() is %d\n", rc);
40     exit(-1);
41   }
42   free(back);
43 }
44
45 // parallelMergesortHelper
46 // -if the level is still > 0, then make parallelMergesortHelper threads
47 // to solve the left and right-hand sides, then merge results
48 // after joining and quit.
49
50 void *parallelMergesortHelper(void *threadarg)
51 {
52   int mid, t, rc;
53   void *status;
54
55   struct thread_data *my_data;
56   my_data = (struct thread_data *) threadarg;
57
58   if (my_data->level <= 0 || my_data->low == my_data->high)
59   {
60     //We have plenty of threads, finish with sequential.
61     mergesortHelper(my_data->lyst, my_data->back,
62         my_data->low, my_data->high);
63     pthread_exit(NULL);
64   }
65
66   //Want joinable threads (usually default).
67   pthread_attr_t attr;
68   pthread_attr_init(&attr);
69   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
70
71   //At this point, we will create threads for the
72   //left and right sides. Must create their data args.
73   struct thread_data thread_data_array[2];
74   mid = (my_data->low + my_data->high)/2;
75
76   for (t = 0; t < 2; t ++)
```

```
77  {
78   thread_data_array[t].lyst = my_data->lyst;
79   thread_data_array[t].back = my_data->back;
80   thread_data_array[t].level = my_data->level - 1;
81  }
82  thread_data_array[0].low = my_data->low;
83  thread_data_array[0].high = mid;
84  thread_data_array[1].low = mid+1;
85  thread_data_array[1].high = my_data->high;
86
87  //Now, instantiate the threads.
88  pthread_t threads[2];
89  for (t = 0; t < 2; t ++)
90  {
91   rc = pthread_create(&threads[t], &attr, parallelMergesortHelper,
92         (void *) &thread_data_array[t]);
93   if (rc)
94   {
95        printf("ERROR; return code from pthread_create() is %d\n", rc);
96        exit(-1);
97       }
98  }
99
100 pthread_attr_destroy(&attr);
101 //Now, join the left and right threads and merge.
102 for (t = 0; t < 2; t ++)
103 {
104  rc = pthread_join(threads[t], &status);
105  if (rc)
106  {
107   printf("ERROR; return code from pthread_join() is %d\n", rc);
108   exit(-1);
109  }
110 }
111
112 //Call the sequential merge now that the left and right
113 //sides are sorted.
114 merge(my_data->lyst, my_data->back, my_data->low, mid, my_data->high);
115
116 pthread_exit(NULL);
117 }
```

Appendix B

Approximate Roots of Polynomials

This appendix summarises the approximate roots of the best coefficient vectors, which maximised the execution times of GSL's polynomial root-finder, obtained from each chapter.

Table B.1: Approximate roots of $17985x^4 - 20437x^3 - 22267x^2 + 15894x + 21508 = 0$ (Chapter 3)

Roots
$1.287408339087713793 \pm 0.380431532870959876i$
$-0.719240421378509631 \pm 0.382468304175276497i$

Table B.2: Approximate roots of $2622x^6 - 6095x^5 - 13873x^4 + 26154x^3 + 28203x^2 - 24011x + 3797 = 0$ (Chapter 3)

Roots
0.235710558547915006
0.403924841403742740
$-1.556879134291467626 \pm 0.392505777534434219i$
$2.399342136070031550 \pm 0.378443939319265454i$

Table B.3: Approximate roots of $20905x^8 - 8575x^7 + -29306x^6 + 4007x^5 - 12774x^4 + 8105x^3 + 23227x^2 + 2291x + 26880 = 0$ (Chapter 3)

Roots
$-0.246274919414626758 \pm 0.846299340317886006i$
$0.287705009833163883 \pm 0.823256707184029879i$
$-1.110474789396435735 \pm 0.241213864506183123i$
$1.274139173983877882 \pm 0.248714743536976562i$

Table B.4: Approximate roots of $-16062x^10 + 8416x^9 + 19083x^8 + 6246x^7 + 6219x^6 - 10917x^5 - 29935x^4 - 23003x^3 + 1673x^2 + 18598x - 15999 = 0$ (Chapter 3)

Roots
$0.095346496125123137 \pm 1.098945088266054748i$
$0.601357465240699440 \pm 0.789554915489905307i$
$0.485372339699980282 \pm 0.377778761256662765i$
$1.330130823575401022 \pm 0.297156580170683582i$
$-1.047507385294156101 \pm 0.292191615961456874i$

Table B.5: Approximate roots of $-13244x^4 + 10440x^3 + 26663x^2 - 11333x - 21492 = 0$ (Chapter 4)

Roots
$-0.905265307193834579 \pm 0.302989136731997732i$
$1.299406050171785898 \pm 0.303729421440160741i$

Table B.6: Approximate roots of $17391x^6 + 32398x^5 - 15982x^4 - 16510x^3 + 28217x^2 - 16035x + 6134 = 0$ (Chapter 4)

Roots
$0.166620896627616477 \pm 0.521038470618695237i$
$0.560725046614891642 \pm 0.322833701144523200i$
$-1.658804743771514545 \pm 0.252771668228436164i$

Appendix C

Execution Times of Benchmarks over Different Problem Sizes

This appendix summarises execution times produced by running a benchmark with different problem input sizes over the different number of threads in Table C.1.

Table C.1: Summary of execution times on different threads and problem size of bubble sort and shell sort

(a) Bubble sort

	Thread(s)		
Problem size	**1**	**2**	**3**
50	142,120	592,020	907,960
100	443,870	1,004,040	1,904,930
200	1,599,750	2,364,490	4,223,610
500	9,343,470	8,548,140	14,131,270
1,000	35,947,990	25,697,090	23,413,880
2,000	141,096,760	86,981,850	72,160,390
5,000	834,296,520	473,509,400	362,390,410

(b) Shell Sort

	Thread(s)		
Problem size	**1**	**2**	**3**
50	11,400	64,740	110,690
100	25,300	77,080	120,810
200	56,480	98,680	157,960
500	166,980	195,830	266,060
1,000	390,540	377,820	443,360
2,000	740,960	658,950	754,740
5,000	1,457,460	1,328,850	1,459,520

Appendix D

Best Genomes of Direct and Indirect Approaches

This appendix presents the best sequences of values (i.e. genomes) generated by meta-heuristics. Particularly, Tables D.1 to D.3 list the best sequences of values for sortings in Chapter 3.

Table D.4 lists the best sequences of values for the polynomial solver. The subdomains, which were constructed from these genomes, delivered the most extreme test inputs for each problem input size in Chapter 4.

Table D.1: Best sequences of values for bubble sort

Arguments	Thread(s)	A sequence of values
100	1	6,502; 30,019; 32,179; 28,217; 26,100; 26,989; 24,507; 29,290; 23,735; 26,975; 5,131; 21,744; 32,528; 13,514; 19,188; 30,833; −4,559; 12,663; 16,999; 19,312; 7,162; 5,074; 3,755; 19,662; 17,843; −1,916; 9,586; 5,298; −75; 12,801; 26,149; 13,989; 21,390; 15,486; 3,259; 7,521; 13,088; 15,063; 848; 5,726; 20,203; 14,879; 8,959; 182; −3,168; −1,050; −13,415; −7,510; −10,064; −4,213; −7,206; −26,082; 2,900; 5,783; 8,622; −1,423; −10,811; −14,620; 4,792; −7,475; −3,270; −10,314; −8,386; −15,544; −2,857; −8,453; −15,959; −12,166; −18,072; −4,637; −7,888; −16,984; −30,131; 1,033; −16,062; −16,033; −16,938; −23,392; −22,374; −23,461; −14,301; −18,646; −24,134; −16,026; −31,305; −27,991; −30,888; −440; −11,090; −7,559; −15,374; −14,796; −30,515; −20,516; −28,384; −23,355; −30,042; −32,457; −26,439 and −32,198
	2	13,337; 5,138; −4,052; −2,960; 13,940; −22,477; −26,957; −6,872; −8,509; −6,761; −5,298; 11,145; 7,151; −17,619; 23,508; 16,805; −20,503; 20,546; −27,020; 22,713; 23,231; −1,738; 28,563; 32,739; 20,768; −15,847; −23,365; 113; 26,407; 30,283; 9,452; 26,235; −5,084; 31,588; 14,724; −28,969; −26,206; 27,604; −11,043; 24,838; 13,880; 10,511; 22,172; 30,619; 24,916; −16,351; 24,074; −7,693; 13,727; −16,963; −3,501; 10,442; −25,331; −298; 24,438; −20,008; 12,337; 14,231; −19,639; 2,844; 5,510; −29,260; −3,169; 357; 8,847; −28,030; −18,993; −1,712; −3,711; 509; −28,532; −26,603; 3,953; −10,410; 28,825; −25,893; 654; −26,807; −10,007; −2,913; −13,099; −24,047; −25,206; −24,154; 8,520; −22,003; −12,503; 7,950; −109; −2,027; −6,943; −2,843; 13,565; −19,340; −1,009; −26,081; −26,076; −7,239; −16,767 and −32,190
	3	30,430; 24,926; 11,767; 25,219; 21,117; 14,261; −1,807; 9,523; 18,402; 28,674; −4,696; 29,700; 2,530; 27,520; −6,645; −112; 24,164; 1,688; 19,629; 13,666; 30,115; 30,280; −25,154; −7,513; 24,902; −18,021; −10,880; 12,642; 5,360; −23,876; 26,382; −9,541; 8,153; 26,067; −7,859; −728; 30,431; −20,497; 9,142; 8,704; −7,526; −5,604; −16,108; 7,840; −16,889; 23,644; −3,805; 30,013; 11,814; −687; −17,301; 19,453; −13,953; 3,031; −15,750; −11,322; −18,208; 21,222; 30,500; 25,909; 21,052; −23,914; 2,329; 945; −18,083; −32,380; 5,829; −10,040; −7,557; −15,567; 10,466; −9,673; −956; −26,312; 22,023; 5,830; −29,593; −28,995; −31,203; 10,273; −27,793; −5,577; −31,385; −27,690; −27,112; −17,297; 12,326; −28,640; −28,568; −27,221; −29,685; −25,141; −26,859; 17,499; −27,024; 20,387; 24,424; 20,376; −24,410 and −32,682
200	1	32,331; 14,012; 29,318; 24,193; 24,665; 25,120; 28,265; 26,726; 31,281; −12,067; 12,913; 26,248; 12,674; 29,889; 21,278; 5,499; 24,173; 12,407; 26,857; 11,942; 23,884; 32,738; 22,702; 16,324; 30,873; 30,028; 12,527; 30,066; 15,246; 27,779; 30,611; 16,656; 20,836; −6,114; 16,036; 15,436; 8,919; 11,203; 26,846; 14,589; 14,809; 30,086; 27,222; 14,276; 12,258; 28,657; 31,215; 26,779; 10,349; 15,395; 5,103; 23,443; −1,983; 18,490; 30,608; 24,117; 20,992; 24,981; 18,132; 25,940; 8,203; 8,520; 8,388; 15,391; 3,314; 18,180; 20,289; 8,867; 18; −558; 8,391; 12,120; −618; −19,757; 2,966; 9,626; 1,162; −7,973; 6,067; 24,808; 28,364; 11,804; 12,312; 2,365; −4,473; −7,700; 8,571; −6,837; 4,528; 7,011; −19,359; −8,463; −6,938; 18,364; −2,352; 28,776; 6,370; 15,044; −5,131; 5,385; 21,437; 28,578; −8,981; −3,442; −29,866; −20,061; −13,925; −19,148; 16,329; −1,300; −11,902; −13,472; −19,678; 5,896; −10,283; −11,221; −19,812; −12,866; −27,127; −5,549; −7,602; 8,592; 8,707; −12,133; 9,976; −2,499; −9,125; −1,388; 12,553; 3,841; −27,104; 5,929; 5,345; 504; 17,636; −5,062; −7,985; −6,359; −19,093; −21,199; −20,207; −23,501; −8,511; −8,316; −4,892; −6,256; −9,848; −7,365; −804; 2,298; −13,716; −15,906; −28,872; 8,884; −15,602; −15,140; 3,321; −26,094; −21,955; −19,649; −29,543; −6,775; −32,645; −18,921; −7,617; −27,421; −14,981; −23,513; −25,825; −23,325; 3,629; −14,174; −21,699; −27,419; −18,563; −25,534; −10,467; −29,159; −9,770; −10,914; −22,246; −79; 4,946; −12,984; −9,419; −10,241; −16,513; −26,727; −13,370; −22,379; −9,564; −23,230; −29,359; −21,086; −8,699; −15,048; −24,904; −27,862; −32,550 and −25,234

Continued on next page...

Continued from previous page...

Arguments	Thread(s)	A sequence of values
200	2	32,398; 13,560; −8,726; 24,683; −5,854; 26,035; −2,617; 28,921; 26,593; 31,896; −14,947; 12,454; −2,889; 23,958; −6,766; 25,766; 29,157; −22,764; 29,674; −1,517; 30,908; 14,612; 100; 18,256; 3,019; 23,600; −6,654; 29,080; −10,670; 18,835; 9,787; −2,764; 12,839; 28,867; 27,398; 24,031; 19,336; −12,283; 30,807; 32,342; 24,109; 31,763; 19,569; 27,458; 8,882; 12,133; −14,901; −27,785; 26,618; 7,093; 3,320; −14,392; −16,359; 182; 11,377; 32,555; 10,805; 18,648; 20,483; 30,624; −18,166; 16,940; 1,035; 4,223; −10,530; 16,459; −18,043; −27,267; −21,889; 1,790; −3,183; −4,983; −19,283; 21,088; 17,909; 27,048; −20,989; 11,432; −15,213; 13,768; 29,100; 21,167; 3,826; 18,321; 30,083; 456; −567; −11,492; 17,380; 23,697; −4,800; −15,385; 16,228; −2,746; −31,584; −25,358; −7,157; 25,521; −21,447; −6,115; 12,500; 23,220; −16,349; −9,770; −21,926; −12,168; −11,847; 27,412; −7,103; −25,894; 8,226; 17,381; −12,339; −9,495; 20,404; −29,067; 6,684; −23,329; 19,237; −29,478; 1,924; −15,237; −16,887; −23,513; 19,664; 14,565; 4,975; 2,387; −8,579; 21,399; −26,186; 27,318; −11,458; 18,116; −31,187; 12,158; −11,871; −12,376; −18,728; 7,118; 11,445; −5,530; 27,388; −19,874; −24,890; −22,725; −21,924; 18,395; −5,209; −22,440; 10,674; −14,127; −2,785; −9,402; 12,599; 1,990; 5,829; 2,634; 15,475; −2,094; −9,440; −15,320; −22,111; −2,138; −661; −6,279; −30,363; −18,695; −9,330; −19,705; −2,217; −14,383; 13,070; −26,171; −7,032; −32,237; −6,171; −537; −15,780; 26,036; 24,956; 9,722; 4,853; −28,791; −26,938; −8,451; −6,587; −9,619; 11,194; −9,999; −24,594; 672; −14,357; −30,764; −27,983; 710; −31,712; −27,937; −3,373 and −32,600
	3	−14,338; 23,437; 7,419; −2,098; −13,220; 32,656; 12,402; 14,392; −26,574; 13,822; −20,515; 15,952; −1,735; −15,812; −12,573; −27,567; −20,875; 4,988; 3,583; 12,809; 23,858; 29,628; −24,858; 1,978; 17,176; −4,176; −8,963; −22,809; −18,562; 5,829; 19,884; 23,245; 16,190; 23,677; 18,767; 28,370; 11,779; 25,090; −20,120; −24,846; 10,052; −15,575; 5,383; −7,060; 9,857; −17,249; −772; −761; 7,428; 10,985; 19,422; 30,424; −10,096; 5,656; −18,265; −21,920; 26,079; −12,600; −6,319; 20,039; −16,630; −19,471; 13,761; 4,283; 16,849; 19,748; −7,162; −25,545; 24,397; −4,055; 7,356; −26,980; 18,098; 1,015; −2,397; 7,701; 11,177; 13,803; 12,567; 6,203; 28,561; 15,005; 1,945; −14,523; −7,114; 5,601; −10,916; 19,426; −11,507; −25,808; −934; 21,706; 29,461; 12,348; 23,361; 19,800; −20,010; 10,755; −19,117; −178; 9,496; −16,454; −24,043; −281; 3,189; 20,854; 11,816; 23,784; 6,317; 10,574; −5,410; 17,888; −22,353; 30,966; 10,603; −9,448; 19,607; −1,736; 22,020; 6,011; −16,407; 20,051; −10,722; 8,574; 11,234; 24,810; 17,709; −5,859; 3,222; −12,480; 12,377; −9,148; −4,368; 2,962; 13,933; 31,829; 16,725; 2,412; −30,382; −2,636; 23,102; −23,227; −23,230; 16,030; 4,556; −21,509; −1,608; −26,739; 5,051; 23,636; −4,223; −25,527; −19,641; −22,460; 12,972; 2,145; 3,526; 2,379; −17,971; −24,822; −27,268; 3,817; −10,023; −29,035; −11,882; −21,768; 6,813; −2,726; −14,188; −16,809; −18,036; −28,588; −2,080; −1,580; 7,388; 9,672; 16,112; 11,982; −30,520; −26,614; 15,850; −12,365; −17,028; −25,096; −13,251; −21,144; 3,828; −27,024; −5,687; 8,494; 26,834; 31,629; 15,255; 9,095; 1,088; 28,020; −25,271; −27,657; 10,768 and −31,528

Table D.2: Best sequences of values for shell sort

Arguments	Thread(s)	A sequence of values
100	1	12811; 5065; 32077; −6566; −9696; 5136; 19578; −15644; −27794; 29715; 31843; 1277; 3862; −9247; 32406; 28384; 5304; 28585; −10971; 21906; −19995; 22399; 2908; 20782; −29089; −14481; 6124; 17624; 1918; −2416; 1565; −19181; 20190; −25707; 3082; 10777; 31061; 3339; 23306; 515; 22529; 26023; 3828; −18776; 26444; 9171; 5918; 5045; 21367; 6421; −30194; 3058; 24017; 29068; 2549; 17378; 6099; 7910; 1089; −2185; −12953; 894; −2207; −21281; −3209; −31358; −3997; 17687; −23440; 23100; 19539; −6401; 11388; −4770; −9533; 11694; −12360; −30696; 3810; −15762; −8999; −30272; −6410; 11842; 2634; −26162; −23857; −5109; −7679; −4846; −10569; −18572; −8246; 31989; −30032; −20631; 263; 15140; −2573 and 6879
	2	32241; −22856; 12366; 19916; 31720; 26080; 10714; 21263; 3968; 25386; −11588; −22084; −6220; −2533; 30334; 8907; −3414; −9771; −22243; 6758; −23909; 25212; 21804; −29912; −27111; −25865; 16358; 11673; 24891; 23320; −21886; 17115; 50; 5093; −23792; 30071; 5404; 6397; 25897; −4249; −7295; −7965; 18437; −22591; 16023; −303; 22104; 26859; −3184; −32054; 6041; −1501; 27511; 27150; −1393; 3093; −30289; 25442; 96; −24980; 32491; −14393; −30317; −31191; −29138; −10771; 25671; 6105; 6851; 18757; −9475; 2336; −25908; 436; −14388; −27666; −28245; 25147; −23873; −5575; 8006; 16977; −4646; 2396; −11126; −27452; −11188; 4189; −18285; −11313; 4151; −28865; 30502; 5357; 15120; 22621; −11095; −4584; −22421 and −32344
	3	25442; −20597; −18299; 15283; 24845; −9915; −4; 14852; 22595; 22562; 11672; −23158; −8696; 14003; 20676; −9996; −26422; −6100; −13904; 15383; −20502; 28202; 26878; 25411; 31892; 13052; 4362; −17060; −9439; −14587; −3230; 31397; 27483; −14100; −18397; 5258; 32363; −25248; 3364; 27582; 9580; −22514; −6678; −13308; −11320; −22136; 18783; 22073; −6020; −11265; −23114; 709; 13196; 3303; 24262; 10259; 23676; 6061; −20073; −32641; 27323; −1212; 31033; −4909; −19862; −2884; 12808; 7307; −30473; −2943; 21568; −19182; −22135; −412; 21934; 17305; −5570; 3297; −25260; 9254; −22178; −25779; 19308; 8840; −25284; 13348; 20028; −31183; 23719; −19572; 28026; 27368; 23189; −17166; 20383; −17852; 21319; −7898; 2061 and −32452
200	1	−19431; −6266; 10617; 5897; −4945; 27077; 11351; 3929; −31960; −16140; −15855; 14061; −14670; −8151; 11928; 19347; 32182; 12204; −28644; 4452; 5453; −21728; −1859; 5699; 10360; −14924; −19472; −5301; −6638; 14128; 8006; −1027; 11744; −20103; 6860; 6485; 21389; −25409; −12927; 19191; −757; 18010; −4375; 4702; 26347; −3165; 9124; 9863; −9255; −12648; −5509; 21589; −21491; 10245; −15035; 29079; 6270; 29595; 2220; 24044; 11353; −30129; −7464; 8802; 8949; −31201; −1726; 9691; −3798; 31836; 10446; −9301; −5788; −9218; −2622; 24985; −8643; −24827; 18810; −30296; 17453; −20327; 21493; −11893; 27639; −21550; 19450; 30267; −10502; 7547; 28706; 8368; −26091; 2432; 3688; 1712; 29225; 28078; −11106; −3786; 20349; −11801; −1581; −15135; −6539; −31014; −18534; −1538; −1093; 410; 20710; 9916; 23511; −24549; −27181; −21801; 14547; −27335; −12457; −3466; −12319; −10116; −3956; 26060; 21069; 18309; −5318; 22809; −4545; −7689; 9830; −1473; −16223; −18114; 7245; −26252; −27180; 11796; −4402; 13984; 19206; 1321; −20807; 944; 20543; −28112; −22527; −5084; −7994; 28173; 29796; −918; −30496; −25425; −29975; −17370; −12491; −28803; −22864; −13346; −12703; 3358; 14141; 8937; 26277; −10023; 28908; 29390; −25475; 7170; 11182; 21111; −32268; −15813; −8242; 15686; −4197; 20795; 23113; −18939; 21103; −2392; −8875; −4779; −4855; 19651; −12160; −12774; 22275; −5806; 9766; −26977; 10464; −30861; −30836; −24894; 1778; −12109; −19752 and −24022

Continued on next page...

Continued from previous page...

Arguments	Thread(s)	A sequence of values
200	2	24974; 2174; 26041; 19854; 10499; −1087; 13327; −15986; 30345; −14702; −16776; −3551; −16246; 8354; 12894; 8291; 22629; 9175; 26676; −32452; 5818; 15221; −7177; −22475; 14007; −19060; 16357; 31599; 21310; 16800; −31297; 15667; 8181; −17782; 1626; −30595; −8294; 15053; −13233; 7338; 191; 8118; −6095; −3880; −30163; −31934; 24674; −7881; 26537; 13929; −32336; −14119; 27918; 11732; 24600; 22857; 5237; −31537; 18550; −17698; −14796; −14101; −24416; −16995; −17343; −4681; −4626; −16351; 5730; 29296; −5939; 30982; 26986; −18877; −24960; −20235; −25263; −11228; −7473; 29887; −12349; −14467; 22577; −7056; −4816; 12679; 6057; −9348; −27443; −13923; −24331; 2498; 17649; 20177; 28041; 19956; 1285; −6727; 10607; −19333; 1833; 30186; −21358; −31007; −28004; −5796; 28995; 19760; 18665; 8731; −2949; −11165; 25612; −19144; −24517; −29212; −25595; −16756; −29197; −12094; 31025; −7683; −28602; −4867; −19467; 4776; 21457; −22906; 2458; −30300; −32448; −28286; −10390; 9219; −21628; 5481; 10408; 29955; 7432; 18867; 5828; −30641; 7764; −8086; −10049; 17003; −22819; 30177; 27255; 4129; −26998; 8270; 23910; −28313; −28948; −9784; −32070; −27533; −10633; 15752; −3570; 24929; 27015; −7179; 3997; 17572; 19439; −8336; 14052; −22918; 18256; −13674; 8018; 11646; 18784; 7977; 14103; 2365; −14731; −20445; −32570; −4014; −277; −19422; 11670; −31181; −17799; 12997; 8682; 22892; 6895; 7236; −9332; −31465; 20291; −29144; −30947; −17537; −25576 and −16659
	3	6431; 12873; 28217; 28068; −6782; 5592; 6025; −4101; −5856; 309; −2848; −9782; −510; 19357; 17412; 24195; 29209; 13562; −3631; 28606; −22192; −11252; −24118; 7539; −6190; 8129; −12453; 30353; 18539; −11023; −4904; 25430; 18662; −20165; 12226; −9975; 21728; −18442; −6469; 29010; −7920; 21255; −586; 24467; 4958; 13176; 19173; −29665; −24645; −27015; −3387; −18028; −29505; −5237; 25654; 30690; 4924; 898; −7269; 27376; −26265; −12407; −10893; 9075; −22864; 10780; 5314; 32475; −4013; −2172; 23363; 22768; 32449; −22536; −30614; −9901; 18558; −11223; −9911; −23425; −11336; 1482; 11581; 14034; 11444; 16109; 30338; 7490; −15251; −15363; −1305; −16666; −19436; 10145; 11934; 25995; 17622; −5925; 25036; 4383; −24607; −17298; −28173; 3714; −14667; −12835; 3795; 24022; 2917; 17195; 26534; 19155; 2321; −30897; −11218; −5828; 8818; −18304; 21672; 2508; 31034; −4715; 22664; 31315; −11630; 28811; 13107; −31866; 28681; −29233; −6845; 5082; 25254; −26862; 20440; 10091; −25396; 22548; 11852; −6343; −11194; −15559; −15217; −7777; −29679; −14224; 26902; 31995; 1334; −20259; −22204; −9601; 31945; −28949; −9556; −2149; −14888; −16901; −27346; −8610; 12494; 14432; 24469; −18374; −8514; 24808; 25657; −27458; 8943; −17928; −19408; −16209; 29542; 14719; 18201; 20483; 12044; 31144; 16041; 19888; −31718; −18686; −22349; −16582; −30236; −18321; −21718; −10747; −6838; −17055; −10106; 12014; 9994; 97; −14441; −11395; −12594; −3272; −22607 and 1085

Table D.3: Best sequences of values for quicksort and merge sort

Sorting	Arguments	A sequence of values
Quicksort	100	−32197; 31817; −31856; −31908; −29633; −31086; −29308; 13256; 9008; −21584; 4642; 23028; −24463; −15716; 10030; −15761; 7685; −15657; −9617; −14161; −16558; −7137; 2502; 22357; −9641; −6164; 12215; −16449; 17298; −759; 21873; 22935; −3851; −17506; 22173; −15680; −11231; 27390; 6546; −1052; 14772; −12340; 14791; 25124; 6637; 30922; 25888; −7395; −18585; 20359; −4955; 24072; −20038; 15775; 2792; −16355; 13240; 25860; 27362; −15881; 14131; −29577; 75; −11217; −17540; 15257; −18026; −24867; 32747; −19695; −15397; 4440; −22952; 10344; 21012; −24880; 4178; −22802; −11113; 29593; 13848; −22853; 10090; 9548; 8243; −8919; 4663; −2930; 19297; 15189; 8655; 11661; 3098; −20721; 19664; 26072; 27351; −28106; 30176 and −32140
	200	−24431; −17273; −32395; −31345; −32511; −32686; 21842; −32566; 18893; −32129; −31364; −7567; −32751; 28692; −31393; −31447; −30005; −30685; −16176; −30910; −32659; −26343; 8699; −17538; 11467; −18973; 7672; −27854; −29638; −24050; −17404; −3112; 20411; −16013; −20649; −726; −32018; −21509; 13995; −5838; 8451; 31892; 11376; 14313; 1163; −17376; −3300; 18980; 6700; −8047; 4228; 21029; −16538; 20523; −4680; 5637; 29265; −16841; 9974; −7285; 27199; 3670; 5969; 6327; 19620; 11873; 25111; 1376; 18310; −11570; 4697; 7568; 5271; 6025; 615; −7280; 21617; 23788; 17245; −27469; −13666; −5578; −18007; −2968; 16561; 18038; −23526; −20648; −21497; −27415; −13402; 4958; 7672; −12743; 3228; 17243; 32748; −4388; 13493; −28459; 9022; −6550; −17470; 6564; −12444; 32767; 23513; 25682; 23516; −3345; −19483; 22617; −21261; −21516; −25738; 12472; 21901; 16123; 17362; −4014; 11771; 16091; −29930; 1275; −23044; 2630; −9619; −1237; −21432; 5606; 4939; 10828; −21118; −29366; −22433; 25319; 32422; 25298; 22595; −26288; −1437; 22321; 4104; −24931; 8198; 24798; −27454; −13409; −11507; −4317; −16679; 2107; 20352; 32716; −225; 23745; −28377; −13550; 27206; −10427; −2666; −22290; 20690; 2171; −17654; 13741; 8726; 21640; 2510; 11026; −27751; −12250; −14749; −7546; −8898; −17595; −17497; 3783; 32654; 24033; 17450; −2512; 32238; −12143; 14909; −2646; −11486; 6350; −27545; −21969; −23494; −5117; 15050; 2365; 9828; 26548; 10447; −27115; 11812 and −25368
Merge sort	100	25831; 12629; 26032; 7010; −304; 18219; −14883; 22391; −8346; −18550; 9488; 24312; −22767; −5563; 24547; −5064; 5606; 11352; −23860; 30818; 32210; −31176; 16792; −25583; 24774; −4147; −4365; −13743; 13283; −13459; −7276; 1930; 18375; −17987; −25697; −21894; −25844; 11324; 28332; 9359; −29111; 1609; 17692; −29459; 30332; 10413; −17889; −23689; −6083; −3815; 18017; −31229; 8796; −25307; 7960; 26615; −1634; 8974; 10663; −5188; −30199; 11057; 32163; 26292; 28791; −1325; 13046; −32210; 8067; −23491; 29510; 29768; 29969; −4596; −21694; 30714; 7262; 11339; −13392; 13462; −7405; −3498; 6114; −12599; 4956; 9560; −13382; 3730; 13076; 21764; 29934; −26915; −30866; 31365; 7371; 5908; 22716; 4435; −6603 and 21625
	200	−10892; 13163; −27018; −13501; 16354; 12603; 5072; −15687; −17665; 18260; −20695; −24298; −13864; 5522; −9302; −16495; −28694; 4321; 3795; 27563; 21622; −11128; −28795; 12279; 569; 24260; 16115; −26234; 757; −23109; −8848; 646; −6436; −24801; 29593; 15245; −11148; −14571; 21246; −24513; 21784; 26245; −19550; −18691; −26094; −32352; 31639; 22694; −26314; −23414; −20068; 11846; 8261; −24091; −14940; −19215; −6318; −22882; −31102; −30939; −18175; −21099; 10683; −23211; −15579; 7361; −7076; 13510; 27967; 294; −4261; −32594; 22871; −22039; −1872; 11976; 31000; 18264; 27371; −8541; −25609; 28399; 32658; −2018; −31076; 20481; 2487; −29573; −15615; −2258; 9489; 4773; −30118; −21129; −29786; 4910; 6793; 19037; −32619; −20926; −18409; −23653; −9286; −23204; 28006; 369; 31487; 3502; −29921; −6027; 32503; 23826; −9839; 3624; 19050; 15763; −32173; 7231; 5153; 21025; −31327; −15698; −2386; 13944; 23503; 16932; −10932; −11741; −12112; 15359; −30823; 23086; −14827; 26848; 9616; −20273; −5138; 8794; −26817; 16873; 10852; −25145; −28205; 18639; −18011; 28008; −10000; 18258; 30673; 12546; 26156; 20808; 22687; −23057; −11840; 17011; 5641; −2186; −30815; 5242; 27612; 14660; 17859; −1500; −22711; −13195; −31785; −31192; 10859; −26617; 6826; 30870; −8819; −28981; 13158; −26934; −31419; 14135; 10978; 2865; 19324; −32230; 12838; −26033; −2259; −8283; 168; −27281; 28201; 23925; 4737; −28811; −23925; 5174; −11331; 5086; 24889; 13759; 14863 and −9513

Table D.4: Best genomes for indirect approach on quartic and sextic equations

Arguments	A sequence of values
5	24296; −28805; 15737; −21498; −14090; −30446; −30000; −239; −25868; −22836; −4472; −11340; 14880; 29357; 18182; −8182; 32140; 9694; 6307; −21380; 26266; −27308; −10590; −15799; −27786; 29326; 16522; −6323; −11815; 28310; 21234; −17185; 27940; 21417; −6602; 24831; −16491; −18368; −415; 3322; 15037; −29968; 9408; 3315; 23832; −11690; −22017; −19237; −28726; −7725; 19597; −2289; 18950; 30564; −19606; −17298; 25921; −9939; 28072; 4583; −19440; −24507; 32199; 26184; 9823; 13765; 32658; 17361; 19028; 30818; −16228; −5419; −13249; 28274; −1502; −6258; −30597; 1611; −15421; −3812; −4266; 10980; −12818; −20632; 3531; −24094; 32730; −25115; 2220; 4307; 15869; −30257; −10419; −28589; 9014; −13411; 18476; 27260; −3735; 4544; 28954; −29206; −19279; −22644; 18648; 24450; −16945; 2871; −25202; −29973; −18429; −5476; 17948; −4142; −28869; −7241; 17633; −28723; −24865; −6182; 22162; −19101; −5558; 1463; 15756; −25956; −28322; 10832; −32647; −31544; 8572; 2018; 30155; 25958; 8120; −16758; 28503; 1571; −2687; −8462; 11538; 16063; −13859; −4983; −23830; 5797; 7790; −11624; −26396; 18676; −27745; 29872; 19539; 30476; −6441; −10839; 30800; −14615; 7797; −27119; 32346; −30768; −23272; 10918; −8811; 4337; −25883; 13850; −32582; −24972; −18312; 26660; 12853; 13150; −26753; 809; −11253; 10437; 408; −26715; −20365; 21198; 12703; 13038; 25422; −13245; −22823; 8446; −23066; −8543; −30082; −8048; 10019; 12191; 26185; 14668; −4235; 29151; 29326; −3610; −31406; 3279; −20952; 29065; −30805; 17657; −18998; 29523; −15427; 3108; 9689; −4462; −3592; −4450; 31301; −13699; −20384; −24228; −16188; −29909; 2299; −32331; 13309; −10291; −18561; −16394; −25929; 20099; 29593; −17848; 14964; 8513; 15146; −3641; 25717; 10051; −21672; 21441; −10613; 15823; 21170; 31942; 16435; −7849; −21338; −13330; 17101; 25899; 9417; −13339; −2473; −18723; −21352; −11941; 10142; −7833; 32413; −9332; −18717; 4040; 13121; 27026; −20491; −22829; 14922; 1465; 21913; −11565; 9077; 15854; −17433; −4674; 29228; 13566; 23224; −4972; −23873; −489; −15744; 29938; 19981; 3901; −30458; 4297; −23731; −3738; 786; −10120; 2696; −9090; −23838; 31277; 22524; −28533; 3800; −32544; −11263; 661; −9309; 12853; 9545; −9019; 11418; 21402; −29845; −385; −9607; 313; 6890; 7692; 17554; 8970; −18953; 15877; 2523; −6721; 17889; 32348; 7146; −31473; −28731; 23554; −18210; 21180; 29447; 11192; −32315; −11229; 9522; −10847; 31945; 20520; 8047; 7573; −8181; 29595; 11970; −22553; −20369; −22261; −32388; 238; 23538; −1861; −11131; 31021; 13097; −9605; 10562; 4826; −20098; 4085; −30223; −11592; −20907; −17277; 22858; −3864; −24794; 15503; −5947; −9256; −27363; 22823; −26852; 19977; 6943; −20076; −7819; −25344; −23974; 1309; −2415; 10712; −6157; 7917; −8495; 20187; 4866; −29667; −11425; −25231; 7657; −19239; 1976; −26890; 12812; −31131; −31104; 19975; −5864; 29885; −22960; 31243; −928; −19133; −19110; 30230; −11585; −3577; 6779; 13222; 10283; 17982; 4120; −7885; −25175; −8698; −26527; 6579; 3893; −12184; −23439; −3581; −27742; −13078; −10636; −15031; 919; 12074; −19295; 32493; 6346; −21964; 4195; 16727; 3495; 5724; −27066; 5671; 30421; 10471; −28315; −31178; 12080; 32560; 20277; −9880; −28528; 1152; −2389; 24807; −2500; −12093; 25385; 3311; −26442; 24068; −13424; −13573; 32692; −17481; −10479; 23425; −3382; −10768; −24963; −12495; −10944; −5064; 22884; 2260; 5848; −23903; −25130; −32411; −18042; 29183; −31494; −15270; −3447; −7346; −26196; −24665; 24650; −12833; 15231; −16038; 4855; −7262; 2655; 6730; 15418; −5469; −1290; 15698; 9602; 28188; 29008; 17803; −13270; 21885; 20543; 5629; −30097; 24167; −7937; 15919; 100; −3817; −18204; −29510; 1367; −14218; −10605; −4750; −10912; 10590; 7924; 32314; 4384; 27325; 13697; 13091; 31588; −7638; 19836; −13977; 27563; 19238; −22743; 2800; −17248; 5309; −2527; −7304; 9880; 941; −1384; 25073; −21789; −10923; 26291; −12730; −614; 6185; −28456; 6569; −22977; −25472; 3074; −3572; 2200; −22371; −4058; −29086; 32363; 21933; −590; −18174; 9848; −21433; −27988; 26098; −13813; −32021; 22583; −26178; −11118; −32239; 17556; 9080; −22235; 27414; 23873; −13835; −10330; 30092; 17402; 32042; −17103; 23835; 12529; −4821; −2502; −11474; −20985; −27731; −3141; −20038; −4519; −22148; 9721; −18332; 13236; −14449; 15697; 29792; −4181; −11273; 14549; 28897; −28721; 25190; −12553; 16191; −2020; 10492; 8883; −9084; −26980; 8724; −22971; −1609; −11075; −1552; −3258; 16599; −1712; −32117; −12324; −4838; 10007; 7640; 6771; −26134; 2785; −32487; −18132; 19537; −12584; −9804; −12622; 8241; −14908; 21530; 13666; 18465; −15826; −8216; −21366; −32530; −17498; −18502; 30241; 26183; 711; 18840; 24456; 18280; 17252; 13469; 18020; −7875; −29903; −19305; −31876; −26437; −11650; −19607; 24853; −24450; −31833; −11465; 13901; −22932; 30137; 12381; −5043; 8702; 21222; 7882; 29092; −4922; 18566; −4884; −16819; 22285; 9655; −1324; −2177; −10979; −1580; 13042; −18965; 19169; −21408; −6636; −21677; 21581; −15129; −15959; 8215; 30124; −25121; 25726; −2226; −3383; 1770; −26566; −9013; −20978; 853; −30395; −5666; −30576; −21043; −2418; 20518; −15232; 1074; −21765; −29477; 11971; 10635; −10321; −18294; −31404; −11792; 3014; −32128; −506; 17113; 30210; 19078; 5223; −20188; −23365; −8135; 23372; −21331; −6219; −19363; −2267; −24372 and 5000

Continued on next page...

Continued from previous page...

Arguments	A sequence of values
7	1359; −26641; 22906; 6123; 24802; 1473; 10671; −26336; −19802; −2743; 3046; −6963; 28884; −16013; 14517; 23227; 9634; −20965; 19951; −26395; 10125; 25109; −30349; −32422; 17872; −3752; −30187; 12790; −3293; −23519; 26343; 20559; −30075; −32677; −21238; −19377; 12996; −29006; 20811; 30588; 13054; −30224; 31082; 7124; −8656; −1917; 28110; −9640; 18572; −29058; 25790; 20313; −19514; −24672; 11956; −11963; 29958; 19156; −24557; −3399; 23862; −16522; 7823; −19408; 17542; 77; −25409; 25731; 17699; −6091; −1454; −1813; −99; −26522; −21591; 24164; 5441; 8557; 31091; 3856; −13851; 2457; −8131; −13312; −28036; −13322; 21097; −4584; 3727; −30890; 10800; −6613; −7400; −25438; 23867; 6531; 21336; 23892; −23245; 29056; 15737; 23928; 11179; −28539; 9884; −10990; −15303; −31226; 19979; −17768; 25229; 24741; 24491; 7631; −7089; −32725; 16908; −29944; 31828; −2047; −764; 5948; −6983; −20235; −19893; 10391; −1352; 4894; −16527; 27653; −4692; −31112; −9654; 5600; 21350; 16986; −21184; 28166; −17036; 7436; −4249; 19220; 16898; −14361; −22841; −23299; 23534; −1968; −28097; 27198; 30558; 6595; 10056; 27100; −29020; −9920; −29444; 18909; −21051; −12818; −29971; 11625; −29657; −9513; 12380; 1397; 13857; −29661; 9418; 18148; 4642; 10104; −10816; 18351; −28788; −716; −21198; 24442; 28465; −6324; 29269; −12183; 16559; 13920; 29877; −26862; −16000; −1857; −13852; −5181; 527; 23387; 14087; −20559; −27342; 20365; −30240; −13277; 18011; 2564; 17617; −13362; 25536; 29880; −1028; −10626; −10018; 29556; −24145; −26970; 7261; −20454; −10732; 25461; −29064; 28222; 5854; 11485; −30735; 28916; 25327; −24487; −5012; −6779; −30749; 11005; −11254; 11468; −30322; −24380; −32122; 8056; −20754; −31572; −13443; −22120; −645; −29852; 6780; 14246; 7426; 12976; −15398; 19934; 7842; −31920; −7989; 12313; 13511; 4345; −25564; −29419; 13741; 4811; −5754; 23655; 3018; 12235; 3899; −16050; 8776; 10602; −2125; −12098; −26327; 11470; 25067; 15716; 26513; −11554; 6800; −9250; −28131; −26196; −6438; −11023; 12481; 30169; −32332; −18613; 22588; −21067; −22787; 3940; −14325; 76; −7128; 25625; 22688; −28860; −14076; −13710; −24069; −22444; 26462; −13520; 13555; −23846; −18498; 30486; −4876; −8768; −16069; −27552; 5025; −7696; −11287; −12801; −13685; 30079; −21832; −20708; 30690; −20470; −5812; 4836; 23910; −5259; −15163; −20518; 15672; −6378; 28176; 29058; 1232; 21146; −21447; −28832; −32108; 16984; −15928; −27532; 10880; 16349; 11092; 7158; 3168; 12952; 29528; −1506; 9883; 3790; −20057; 7585; 26960; 21520; −15966; 3954; −3172; −14340; −12249; 19332; −26749; −22001; −23285; −21348; 23380; −10876; 17996; −7465; −29474; −21259; 21135; −23836; −11438; −19290; 31936; 24306; −31583; 2535; −9325; 18869; −31557; −21522; −3713; 32379; −13546; 14675; −1760; −7510; 18260; −18427; 5975; −10076; 28028; −16584; 28217; 8981; 6374; 21204; −16511; −22590; 21190; −583; −15982; 1151; 1592; −11708; −21380; 32396; 11360; 18760; −5872; 1578; −346; −20819; 22900; 17391; −32541; −6016; −10281; −28359; 10741; −14477; 14416; −24394; −15723; −12774; −29063; −15426; −18969; 21196; 29677; −32720; 22258; −4919; 7104; −9094; 31561; −20692; −25037; 30693; −13399; −10614; 18736; 3089; −20169; −7831; −26104; −20280; 4200; −18936; −4332; −20779; 11305; −15163; −30183; 20365; −19605; −7061; −26966; −13782; 12671; 18309; −25157; −31769; −12387; −21241; 14816; 7366; 21858; −22905; −21485; −21853; 20282; 4934; −8244; −32528; 28716; 185; 11861; 13700; −9979; 547; 7895; −1951; −15929; 32017; −1080; 6493; 19023; −508; 20096; 18894; 6394; −21872; 32146; −11734; 29651; 10946; −9654; 21823; −558; 27386; −16528; 4644; −32446; 19273; 32199; 8058; 8260; −22004; −17608; −15493; 25078; 16359; 28685 and −32142

Abbreviations

The following is a list of abbreviations and acronyms used throughout the thesis. The page on which each one is defined or first used is also given.

AMP Asymmetric Multiprocessing . . . 21

ASIC Application-Specific Integrated Circuit . . . 21

AWS Amazon Web Services . . . 184

BCET Best-Case Execution Time . . . 31

BMP Bound Multiprocessing . . . 21

CISC Complex Instruction Set Computer . . . 15

COM Computer-on-Module . . . 62

DSP Digital Signal Processing . . . 18

EA Evolutionary Algorithm . . . 40

EC Evolutionary Computation . . . 40

ECJ Java-based Evolutionary Computation Research System . . . 59

ES Evolution Strategy . . . 40

FTP File Transfer Protocol . . . 64

GA Genetic Algorithm . . . 2

GCC GNU Compiler Collection . . . 62

GCJ GNU Compiler for Java . . . 62

glibc GNU C library . . . 68

GNU GNU operating system . . . 66

GP Genetic Programming . . . 40

GSL GNU Scientific Library . . . 65

HC Stochastic Hill Climbing . . . 6

ILP Instruction-Level Parallelism . . . 9

IPC Inter-Processor Communication . . . 21

IPR Intellectual Property Right . . . 26

IQR Interquartile Range . . . 87

ISA Instruction Set Architecture . . . 15

JRE Java Runtime Environment . . . 62

JVM Java Virtual Machine . . . 186

MIMD Multiple Instructions, Multiple Data . . . 12

MISD Multiple Instructions, Single Data . . . 12

MPSoC Multiprocessor System-on-Chip . . . 13

MSI Modified, Shared and Invalid protocol . . . 3

NoC Network-on-Chip . . . 20

OS Operating System . . . 2

PCI Peripheral Component Interconnect . . . 62

PRNG Pseudorandom Number Generator . . . 80

QoS Quality of Service . . . 47

RISC Reduced Instruction Set Computer . . . 15

RS Random Search . . . 34

RTOS Real-Time Operating System . . . 24

SA Simulated Annealing . . . 4

SBC Single Board Computer . . . 62

SBSE Search-Based Software Engineering . . . 4

SBST Search-Based Software Testing . . . 2

SHC Steepest Ascent Hill Climbing . . . 6

SIMD Single Instruction, Multiple Data . . . 12

SISD Single Instruction, Single Data . . . 11

SMP Symmetric Multiprocessing 21

SSH Secure Shell 64

SUT Software Under Test 5

SWaP Space, Weight and Power 10

TLP Thread-Level Parallelism 12

TS Tabu Search 34

VLIW Very Long Instruction Word 12

WCET Worst-Case Execution Time 1

WCRT Worst-Case Response Time 29

References

[1] K. Srivisut, J. A. Clark, and R. F. Paige, "Search-based temporal testing in an embedded multicore platform," in *Proceedings of 21st International Conference on Applications of Evolutionary Computation*, Parma, Italy, Apr. 4–6, 2018, pp. 794–809, DOI: 10.1007/978-3-319-77538-8_53.

[2] ——, "Dependent input sampling strategies: Using metaheuristics for generating parameterised random sampling regimes," in *Proceedings of the Genetic and Evolutionary Computation Conference*, Kyoto, Japan, Jul. 15–19, 2018, pp. 1451–1458, DOI: 10.1145/3205455.3205495.

[3] A. Burns and J. A. McDermid, "Real-time safety-critical systems: Analysis and synthesis," *Software Engineering Journal*, vol. 9, no. 6, pp. 267–281, Nov. 1994, DOI: 10.1049/sej.1994.0036.

[4] A. Engel, *Verification, Validation and Testing of Engineered Systems*, ser. Wiley Series in Systems Engineering and Management, A. P. Sage, Ed. John Wiley & Sons, Inc., 2010.

[5] H. Pohlheim and J. Wegener, "Testing the temporal behavior of real-time software modules using extended evolutionary algorithms," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation*, Orlando, Florida, Jul. 13–17, 1999, p. 1795. [Online]. Available: http://dl.acm.org/citation.cfm?id=2934046.2934210

[6] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems," *The Computer Journal*, vol. 57, no. 5, pp. 759–774, May 2014, DOI: 10.1093/comjnl/bxt027.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschu-

lat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, May 2008, DOI: 10.1145/1347375.1347389.

[8] F. Mueller and J. Wegener, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," in *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, Jun. 3–5, 1998, p. 144, DOI: 10.1109/RTTAS.1998.683198.

[9] P. McMinn, "Search-based software testing: Past, present and future," in *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Berlin, Germany, Mar. 21–25, 2011, pp. 153–163, DOI: 10.1109/ICSTW.2011.100.

[10] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology*, vol. 51, no. 6, pp. 957–976, Jun. 2009, DOI: 10.1016/j.infsof.2008.12.005.

[11] T. Kelter, H. Borghorst, and P. Marwedel, "Wcet-aware scheduling optimizations for multi-core real-time systems," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Agios Konstantinos, Greece, Jul. 14–17, 2014, pp. 67–74, DOI: 10.1109/SAMOS.2014.6893196.

[12] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Amsterdam, Netherlands, Oct. 4–9, 2015, pp. 220–229, DOI: 10.1109/CODESISSS.2015.7331385.

[13] S. Poulding and J. A. Clark, "The problems with multi-core: Challenges for the critical systems community," Department of Computer Science, University of York, York YO10 5GH, UK, Tech. Rep., Dec. 2012, version 0.1.

[14] R. Fuchsen, "How to address certification for multi-core based ima platforms: Current status and potential solutions," in *Proceedings of the 29th Digital Avionics Systems Conference*, Salt Lake City, Utah, USA, Oct. 3–7, 2010, pp. 5.E.3–1–5.E.3–11, DOI: 10.1109/DASC.2010.5655461.

[15] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, Graz, Austria, Apr. 13–17, 2015, pp. 1–12, DOI: 10.1109/ICST.2015.7102580.

[16] Y. Zhang, M. Harman, and A. Mansouri, "The sbse repository: a repository and analysis of authors and research articles on search based software engineering," 2017, Accessed on: Jul. 8, 2017. [Online]. Available: http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

[17] U. Khan and I. Bate, "Wcet analysis of modern processors using multi-criteria optimisation," in *Proceedings of the 1st International Symposium on Search Based Software Engineering*, Windsor, UK, May 13–15, 2009, pp. 103–112, DOI: 10.1109/SSBSE.2009.20.

[18] I. Bate and U. Khan, "Wcet analysis of modern processors using multi-criteria optimisation," *Empirical Software Engineering*, vol. 16, no. 1, pp. 5–28, Feb. 2011, DOI: 10.1007/s10664-010-9133-9.

[19] G. E. Moore, "Cramming more components onto integrated circuits," in *Readings in Computer Architecture*, M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[20] Y. Chen, C. Chakrabarti, S. Bhattacharyya, and B. Bougard, "Signal processing on platforms with multiple cores: Part 1—overview and methodologies [from the guest editors]," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 24–25, Nov. 2009, DOI: 10.1109/MSP.2009.934556.

[21] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, Nov. 2009, DOI: 10.1109/MSP.2009.934110.

[22] F. J. Pollack, "New microarchitecture challenges in the coming generations of cmos process technologies," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, Haifa, Israel, Nov. 16–18, 1999, p. 2. [Online]. Available: http://dl.acm.org/citation.cfm?id=320080.320082

[23] L. Karam, I. Alkamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in multicore dsp platforms," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 38–49, Nov. 2009, DOI: 10.1109/MSP.2009.934113.

[24] B. Barney, "Introduction to parallel computing," 2017, Accessed on: Jul. 8, 2017. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/

[25] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972, DOI: 10.1109/TC.1972.5009071.

[26] S. Akhter and J. Roberts, *Multi-Core Programming: Increasing Performance Through Software Multi-threading*, D. J. Clark, Ed. Intel Press, Apr. 2006.

[27] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "Ilp versus tlp on smt," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, Portland, Oregon, USA, Nov. 13–19, 1999, pp. 1–10, DOI: 10.1109/SC.1999.10050.

[28] *QorIQ® P4 Series - Fact Sheet (REV 4)*, NXP Semiconductors, Apr. 2014.

[29] I. Richardson and P. Ballantyne, "An overview of h.264 advanced video coding," 2017, Accessed on: Jul. 8, 2017. [Online]. Available: https://www.vcodex.com/an-overview-of-h264-advanced-video-coding/

[30] G.-P. D. Musumeci and M. Loukides, *System Performance Tuning*, 2nd ed., M. Loukides, Ed. O'Reilly Media, Inc., Feb. 2002.

[31] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc., 2013.

[32] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design, T. Green and N. McFadden, Eds. Morgan Kaufmann Publishers, Inc., Sep. 2011.

[33] R. J. Leng, "The secrets of pc memory: Part 1," Nov. 2007, Accessed on: Jul. 8, 2017. [Online]. Available: https://www.bit-tech.net/reviews/tech/memory/the_secrets_of_pc_memory_part_1/3/

[34] "Running amp, smp, or bmp mode for multicore embedded systems," Freescale Semiconductor, Inc., Tech. Rep., 2012. [Online]. Available: https://www.nxp.com

[35] *QNX® Neutrino® RTOS System Architecture*, QNX Software Systems Limited, Feb. 2014.

[36] "Embedded multicore: An introduction," Freescale Semiconductor, Inc., Tech. Rep., Jul. 2009. [Online]. Available: http://www.freescale.com

[37] A. Vajda, "Multi-core and many-core processor architectures," in *Programming Many-Core Chips*. Springer US, 2011, pp. 9–43, DOI: 10.1007/978-1-4419-9739-5_2.

[38] S. Poulding and J. A. Clark, "Strategies for multicore: Verification and certification in high-integrity systems," Department of Computer Science, University of York, York YO10 5GH, UK, Tech. Rep., Jun. 2013, version 1.0.

[39] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. Cambridge University Press, 2008.

[40] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, DOI: 110.1016/j.jss.2013.02.061.

[41] *829-2008—IEEE Standard for Software and System Test Documentation*, IEEE Std., Jul. 2008, DOI: 0.1109/IEEESTD.2008.4578383.

[42] B. F. Oladejo and D. T. Ogunbiyi, "An empirical study on the effectiveness of automated test case generation techniques," *American Journal of Software Engineering and Applications*, vol. 3, no. 6, pp. 95–101, Dec. 2014, DOI: 10.11648/j.ajsea.20140306.15.

[43] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, Dec. 1997, DOI: 10.1145/267580.267590.

[44] N. R. Storey, *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[45] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Software Testing, Verification & Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004, DOI: 10.1002/stvr.v14:2.

[46] A. Burns, "Preemptive priority-based scheduling: An appropriate engineering approach," in *Advances in Real-time Systems*, S. H. Son, Ed. Prentice-Hall, Inc., 1995, pp. 225–248. [Online]. Available: http://dl.acm.org/citation.cfm?id=207721.207731

[47] N. C. Audsley, I. J. Bate, and A. Burns, "Putting fixed priority scheduling theory into engineering practice for safety critical applications," in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Brookline, MA, USA, Jun. 10–12, 1996, pp. 2–10, DOI: 10.1109/RTTAS.1996.509517.

[48] *ISO/IEC 25010:2011—Systems and Software engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*, ISO Std., Mar. 2011. [Online]. Available: https://www.iso.org/standard/35733.html

[49] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science, B. Meyer and M. Nordio, Eds. Springer Berlin Heidelberg, 2012, vol. 7007, pp. 1–59, DOI: 10.1007/978-3-642-25231-0_1.

[50] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, Jul. 2013, DOI: 10.1016/j.ins.2013.02.041.

[51] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, Oct. 2015, version 2.2. [Online]. Available: https://cs.gmu.edu/~sean/book/metaheuristics/

[52] A. Kheiri, "Multi-stage hyper-heuristics for optimisation problems," PhD thesis, School of Computer Science, The University of Nottingham, Nottingham NG8 1BB, UK, Dec. 2014. [Online]. Available: http://eprints.nottingham.ac.uk/29960/1/main.pdf

[53] M. Birattari, L. Paquete, T. Stützle, and K. Varrentrapp, "Classification of metaheuristics and design of experiments for the analysis of components," Darmstadt University of Technology, Darmstadt, Germany, Tech. Rep., Nov. 2001.

[54] E. Aarts and J. K. Lenstra, Eds., *Local Search in Combinatorial Optimization*, 1st ed. John Wiley & Sons, Inc., 1997.

[55] E.-G. Talbi, *Metaheuristics: From Design to Implementation.* John Wiley & Sons, Inc., Jun. 2009.

[56] K.-L. Du and M. Swamy, *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature*, 1st ed. Birkhäuser Basel, 2016, DOI: 10.1007/978-3-319-41192-7.

[57] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer Publishing Company, Inc., 2007.

[58] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983, DOI: 10.1126/science.220.4598.671.

[59] G. Dueck and T. Scheuer, "Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing," *Journal of Computational Physics*, vol. 90, no. 1, pp. 161–175, Sep. 1990, DOI: 10.1016/0021-9991(90)90201-B.

[60] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: a survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, Dec. 2013, DOI: 10.1057/jors.2013.71.

[61] P. Ross, "Hyper-heuristics," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Springer US, 2005, pp. 529–556, DOI: 10.1007/0-387-28356-0_17.

[62] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A classification of hyper-heuristic approaches," in *Handbook of Metaheuristics*, 2nd ed., ser. International Series in Operations Research & Management Science, M. Gendreau and J.-Y. Potvin, Eds. Springer US, 2010, vol. 146, pp. 449–468, DOI: 10.1007/978-1-4419-1665-5_15.

[63] N. Pillay, "A review of hyper-heuristics for educational timetabling," *Annals of Operations Research*, vol. 239, no. 1, pp. 3–38, Apr. 2016, DOI: 0.1007/s10479-014-1688-1.

[64] E. Özcan, B. Bilgin, and E. E. Korkmaz, "A comprehensive analysis of hyper-heuristics," *Intelligent Data Analysis*, vol. 12, no. 1, pp. 3–23, Jan. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1368027.1368029

[65] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones, "Systematic testing of real-time systems," in *Proceedings of the 4th International Conference on Software Testing Analysis and Review*, Amsterdam, Netherlands, Dec. 2–6, 1996.

[66] 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, Jun. 1997, DOI: 10.1023/A:1018551716639.

[67] J. T. Alander, T. Mantere, G. Moghadampour, and J. Matila, "Searching protection relay response time extremes using genetic algorithm-software quality by optimization," in *Proceedings of the 4th International Conference on Advances in Power System Control, Operation and Management*, Hong Kong, Nov. 11–14, 1997, pp. 95–99, DOI: 10.1049/cp:19971811.

[68] N. Tracey, J. A. Clark, and K. Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *Proceedings of the IFIP International Workshop on Dependable Computing and Its Applications*, Johannesburg, South Africa, Jan. 12–14, 1998.

[69] M. O'Sullivan, S. Vössner, and J. Wegener, "Testing temporal correctness of real-time systems—a new approach using genetic algorithms and cluster analysis," in *Proceedings of the 6th European Conference on Software Testing, Analysis and Review*, Munich, Germany, Nov. 30–Dec. 4, 1998.

[70] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, no. 3, pp. 275–298, Nov. 1998, DOI: 10.1023/A:1008096431840.

[71] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 2–4, 1998, pp. 134–143, DOI: 10.1109/REAL.1998.739738.

[72] H. G. Groß, B. F. Jones, and D. E. Eyres, "Structural performance measure of evol-

utionary testing applied to worst-case timing of real-time systems," *IEE Proceedings - Software*, vol. 147, no. 2, pp. 25–30, Apr. 2000, DOI: 10.1049/ip-sen:20000525.

[73] J. Wegener, R. Pitschinetz, and H. Sthamer, "Automated testing of real-time tasks," in *Proceedings of the 1st International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, Jun. 4–5, 2000.

[74] H. Groß, "A prediction system for dynamic optimisation-based execution time analysis," in *Proceedings of the 1st International Workshop on Software Engineering using Metaheuristic Innovative Algorithms*, Toronto, Canada, May 14, 2001.

[75] J. Wegener and F. Mueller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, Nov. 2001, DOI: 10.1023/A:1011132221066.

[76] H. Groß, "An evaluation of dynamic, optimisation-based worst-case execution time analysis," in *Proceedings of the 1st International Conference on Information Technology: Prospects and Challenges in the 21st Century*, Kathmandu, Nepal, May 23–26, 2003, pp. 1–7.

[77] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, Washington DC, USA, Jun. 25–29, 2005, pp. 1021–1028, DOI: 10.1145/1068009.1068183.

[78] M. Tlili, S. Wappler, and H. Sthamer, "Improving evolutionary real-time testing," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Seattle, Washington, USA, Jul. 8–12, 2006, pp. 1917–1924, DOI: 10.1145/1143997.1144316.

[79] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, Brussels, Belgium, Jul. 6, 2010, pp. 136–146, DOI: 10.4230/OASIcs.WCET.2010.136.

[80] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, Jun. 1997, DOI: 10.1145/268806.268810.

[81] S. Luke, "Ecj23: a java-based evolutionary computation research system," 2015, Accessed on: Mar. 29, 2017. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[82] ——, "Ecj-interest discussion list," 2017, Accessed on: Jul. 8, 2017. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[83] D. R. White, "Software review: the ecj toolkit," *Genetic Programming and Evolvable Machines*, vol. 13, no. 1, pp. 65–67, Mar. 2012, DOI: 10.1007/s10710-011-9148-z.

[84] S. Luke, "Ecj then and now," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Berlin, Germany, Jul. 15–19, 2017, pp. 1223–1230, DOI: 10.1145/3067695.3082467.

[85] ——, *The ECJ Owners Manual: A User Manual for the ECJ Evolutionary Computation Library*, 2015. [Online]. Available: https://cs.gmu.edu/~eclab/projects/ecj/

[86] D. Abramson, M. Krishnamoorthy, and H. Dang, "Simulated annealing cooling schedules for the school timetabling problem," *Asia-Pacific Journal of Operational Research*, vol. 16, no. 1, May 1999.

[87] M. Levy and T. M. Conte, "Embedded multicore processors and systems," *IEEE Micro*, vol. 29, no. 3, pp. 7–9, May 2009, DOI: 10.1109/MM.2009.41.

[88] C. E. Howard, "Modern microprocessors: Robust, high-performance aerospace and defense systems harness the power of innovative microprocessors," Mar. 2012, Accessed on: Jul. 8, 2017. [Online]. Available: http://www.militaryaerospace.com/articles/print/volume-23/issue-3/technology-focus/modern-microprocessors.html/

[89] "Qoriq development systems with com express® modules: Ease the 'make vs. buy' decision for oems," Freescale Semiconductor, Inc., Tech. Rep., 2010.

[90] "Comx-p4080 com express module: Installation and use," Artesyn Embedded Technologies, Inc., Tech. Rep., Jul. 2016.

[91] "P4080 development system user's guide," Freescale Semiconductor, Inc., Tech. Rep., Jul. 2011.

[92] "Xcalibur1600—freescale eight-core p4080 processor-based conduction- or air-cooled 6u cpci module," Data Sheet, Extreme Engineering Solutions, Inc., Tech. Rep., 2016. [Online]. Available: https://www.xes-inc.com/products/end-of-life-sbcs/xcalibur1600/

[93] J. Law and G. Pfeifer, "Gcc, the gnu compiler collection," 2017, Accessed on: Aug. 25, 2017. [Online]. Available: https://gcc.gnu.org

[94] "Comx-p4040 & comx-p4080—high performance nxp qoriq modules," Data Sheet, Artesyn Embedded Technologies, Inc., Tech. Rep., Aug. 2017.

[95] B. A. Forouzan, *TCP/IP Protocol Suite*, 4th ed. McGraw-Hill, 2010.

[96] C. M. Lonvick and T. Ylonen, "The secure shell (ssh) protocol architecture," RFC Editor, Jan. 2006, DOI: 10.17487/RFC4251.

[97] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series, B. W. Kernighan, Ed. Addison-Wesley Longman Publishing Co., Inc., Mar. 2009.

[98] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, *GNU Scientific Library Reference Manual*, 2nd ed., Network Theory Ltd., Nov. 2015, for GSL Version 2.1.

[99] M. M. Andersen, B. Barker, S. Brisard, A. D. Chou, M. Diggory, R. B. Donkin, R. B. Donkin, O. Ertl, L. Maisonobe, T. O'Brien, J. Pietschmann, D. Pourbaix, G. Sadowski, G. Sterijevski, B. Worden, and E. Ward, "Commons math: the apache commons mathematics library," 2016, Accessed on: Mar. 29, 2017. [Online]. Available: http://commons.apache.org/proper/commons-math/

[100] M. T. Flanagan, "Michael thomas flanagan's java scientific library," 2015, Accessed on: Mar. 29, 2017. [Online]. Available: http://www.ee.ucl.ac.uk/~mflanaga/java/index.html

[101] L. Chao, *Networking Systems Design and Development*. CRC Press, 2010.

[102] S. Chandrasekaran, M. Gu, J. Xia, and J. Zhu, "A fast qr algorithm for companion matrices," in *Recent Advances in Matrix and Operator Theory*, ser. Operator Theory:

Advances and Applications, J. A. Ball, Y. Eidelman, J. W. Helton, V. Olshevsky, and J. Rovnyak, Eds. Birkhäuser Basel, 2008, vol. 179, pp. 111–143.

[103] V. Y. Pan and A.-L. Zheng, "New progress in real and complex polynomial root-finding," *Computers & Mathematics with Applications*, vol. 61, no. 5, pp. 1305–1334, Mar. 2011, DOI: 10.1016/j.camwa.2010.12.070.

[104] A. Edelman and H. Murakami, "Polynomial roots from companion matrix eigenvalues," *Mathematics of Computation*, vol. 64, no. 210, pp. 763–776, Apr. 1995, DOI: 10.2307/2153450.

[105] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

[106] G. Dahlquist and Å. Björck, *Numerical Methods in Scientific Computing*. Society for Industrial and Applied Mathematics, Apr. 2008, vol. 2.

[107] E. Jarlebring, "Numerical linear algebra (sf3580)," Lecture notes, Department of Mathematics, KTH Royal Institute of Technology, 2014, Accessed on: Feb. 9, 2018. [Online]. Available: https://people.kth.se/~eliasj/

[108] C. Breshears, *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, 1st ed. O'Reilly Media, Inc., May 2009.

[109] C. O'Donell, K. McMartin, S. Poyarekar, and M. Frysinger, *The GNU C Library Manual*, 2017, Accessed on: Mar. 29, 2017. [Online]. Available: https://www.gnu.org/software/libc/manual/

[110] J. Sparger, "Cosc 462: Parallel programming," Department of Electrical Engineering & Computer Science at the University of Tennessee, Knoxville, Accessed on: Jul. 9, 2017. [Online]. Available: http://web.eecs.utk.edu/~jsparger/cosc462/pp/parallelProject3/

[111] B. Weems, "Cse 4351: Parallel processing," Department of Computer Science and Engineering at the University of Texas, Arlington, Accessed on: Jul. 9, 2017. [Online]. Available: http://ranger.uta.edu/~weems/NOTES4351/cse4351.html

[112] J. Stough, "Strategies for introducing parallelism with python," Department of Computer Science at Washington and Lee University, Lexington, Accessed on: Jul. 9, 2017. [Online]. Available: http://home.wlu.edu/~stoughj/SC13/

[113] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Feb. 2003.

[114] P. Pacheco, *An Introduction to Parallel Programming*, T. Green and N. McFadden, Eds. Morgan Kaufmann Publishers Inc., 2011, DOI: 10.1016/B978-0-12-374260-5.00007-5.

[115] D. E. Knuth, *Sorting and Searching*, 2nd ed., ser. The Art of Computer Programming. Addison Wesley Longman Publishing Co., Inc., 1998, vol. 3.

[116] R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed. Sams Publishing, Dec. 2002.

[117] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[118] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, Jul. 1961, DOI: 10.1145/366622.366644.

[119] B. Weems, "Parallel processing (cse 4351)," Lecture notes, Department of Computer Science and Engineering, University of Texas at Arlington, 2001, Accessed on: Feb. 9, 2018. [Online]. Available: http://ranger.uta.edu/~weems/

[120] J. J. McConnell, *Analysis of Algorithms: An Active Learning Approach*, 2nd ed. Jones and Bartlett Publishers, Inc., 2007.

[121] H. Erkiö, "A heuristic approximation of the worst case of shellsort," *BIT Numerical Mathematics*, vol. 20, no. 2, pp. 130–136, Jun. 1980, DOI: 10.1007/BF01933185.

[122] M. Birattari, *Tuning Metaheuristics: A Machine Learning Perspective*, 1st ed., ser. Studies in Computational Intelligence. Springer-Verlag Berlin Heidelberg, 2009, vol. 197, DOI: 10.1007/978-3-642-00483-4.

[123] A. M. Law, *Simulation Modeling and Analysis*, 5th ed. McGraw-Hill Education, 2015.

[124] R. S. Francis and I. D. Mathieson, "A benchmark parallel sort for shared memory multiprocessors," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1619–1626, Dec. 1988, DOI: 10.1109/12.9738.

[125] N. Pillay and D. Beckedahl, "Evohyp—a java toolkit for evolutionary algorithm hyper-heuristics," in *Proceedings of the IEEE Congress on Evolutionary Computation*, San Sebastian, Spain, Jun. 5–8, 2017, DOI: 10.1109/CEC.2017.7969636.

[126] G. Ochoa, M. Hyde, T. Curtois, J. Vazquez-Rodriguez, J. Walker, M. Gendreau, G. Kendall, B. McCollum, A. Parkes, S. Petrovic, and E. Burke, "Hyflex: A benchmark framework for cross-domain heuristic search," in *Proceedings of the 12th European Conference on Evolutionary Computation in Combinatorial Optimization*, Malaga, Spain, Apr. 11–13, 2012, pp. 136–147, DOI: 10.1007/978-3-642-29124-1_12.

[127] J. Swan, E. Özcan, and G. Kendall, "Hyperion—a recursive hyper-heuristic framework," in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science, C. A. C. Coello, Ed. Springer Berlin Heidelberg, 2011, vol. 6683, pp. 616–630, DOI: 10.1007/978-3-642-25566-3_48.

[128] N. Pillay and D. Beckedahl, *EvoHyp: A Java Library for Evolutionary Algorithm Hyper-Heuristics Version 1.0*, University of KwaZulu-Natal, Mar. 2017. [Online]. Available: http://titancs.ukzn.ac.za/EvoHyp.aspx

[129] M. Galassi and J. Theiler, "Gsl—gnu scientific library," 2016, Accessed on: Mar. 29, 2017. [Online]. Available: https://www.gnu.org/software/gsl/