GENETIC PROGRAMMING FOR LOW-RESOURCE SYSTEMS

DAVID R. WHITE

PhD Thesis

University of York
Department of Computer Science

March 2010

David R. White: *Genetic Programming for Low-Resource Systems,* PhD Thesis, © March 2010

For my family.

ABSTRACT

Embedded systems dominate the computing landscape. This dominance is increasing with the advent of ubiquitous computing whereby lightweight, low-resource systems are being deployed on a vast scale. These systems present new engineering challenges: high-volume production places a stronger emphasis on absolute cost, resources available to executing software are highly constrained, and physical manufacturing capabilities approach hard limits. Add to this the sensitive nature of many of these systems, such as smartcards used for financial transactions, and the development of these systems becomes a formidable engineering challenge.

For the software engineer, the incentive to produce efficient and resource-aware software for these platforms is great, yet existing tools do not support them well in this task. It is difficult to assess the impact of decisions made at the source code level in terms of how they change a system's resource consumption. Existing toolchains, together with the very complex interactions of software and their host processors, can produce unforeseen implications at run-time of even small changes.

We could describe such a situation as an instance of *programming the unprogrammable*, and Genetic Programming is one solution method used for such problems. Genetic Programming, inspired by nature's ability to solve problems involving complex interactions and strong pressures on resource consumption, is a clear candidate for attacking the challenges presented in these systems. Genetic Programming facilitates the creation and manipulation of source code in a way that grants us fine control over its measurable characteristics.

In this thesis, I investigate the potential of Genetic Programming as a tool in controlling the non-functional properties of software, as a new method of designing code for low-resource systems. I demonstrate the feasibility of this approach, and investigate some of the ways Genetic Programming could be utilised by a practitioner. In doing so, I also identify key components that any application of Genetic Programming to such a domain will require.

I review current low-resource system optimisation, Genetic Programming and methods for simultaneously handling multiple requirements. I present a series of empirical investigations designed to provide evidence for and against a set of hypotheses regarding the success of Genetic Programming in solving problems within the low-resource systems domain. These experiments include the creation of new software, the improvement of existing software and the fine-grained control of resource usage in general.

To conclude, I review the progress made, reassess my hypotheses, and outline how these new methods can be carried forward to a wide range of applications.

DECLARATION

I declare that all the work in this thesis is my own, except where attributed to another author. Some ideas and figures have appeared in the following publications:

David R. White and Simon Poulding. A Rigorous Evaluation of Crossover and Mutation in Genetic Programming.

Proceedings: EuroGP 2009.

This paper [White and Poulding, 2009] is a demonstration of how to execute rigorous empirical work when working with highly parameterised evolutionary algorithms. The methods explored in this paper are subsequently used in Chapter 7 to investigate the importance of certain parameters in the framework proposed there.

Andrea Arcuri, David R. White, John A. Clark, Xin Yao. Multi-Objective Improvement of Software using Co-evolution and Smart Seeding.

Proceedings: SEAL 2008.

This paper [Arcuri et al., 2008] is the foundation of Chapter 7. It outlines the framework used in that chapter, and includes two of the case studies reported in the chapter. It answers some of the questions investigated in Chapter 7, whilst the rest are in the paper described below.

David R. White, John A. Clark, Jeremy Jacob, Simon Poulding. Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators.

Proceedings: GECCO 2008.

This paper [White et al., 2008] corresponds closely to the work in Chapter 6.

David R. White, Juan M. E. Tapiador, Julio Cesar Hernandez-Castro and John A. Clark. Fine-Grained Timing using Genetic Programming.

Proceedings: EuroGP 2010.

This paper contains an abridged version of the work in Chapter 8, with the same experimental results as reported therein.

The following also contain ideas and figures within this thesis and are currently under review:

David R. White, Andrea Arcuri and John A. Clark. Evolutionary Improvement of Programs.

Submitted for journal review.

This paper contains the work from Chapter 7 that was not published at SEAL 2008. It has further case studies and more detailed empirical exploration of the framework proposed.

ACKNOWLEDGMENTS

First and foremost: acknowledgments to John A. Clark for his support, encouragement and guidance. Thanks to Jeremy Jacob for making me think more rigorously and suggesting new lines of thought, and to Susan Stepney for all her feedback, encouragement and insightful observations. To Juan and Julio for the most enjoyable and exhausting creative discussions I have encountered, with an extra thank you to Juan for all the help and time he has given me during the last few months of the thesis, to be repaid in full at Pivo.

Huge thanks to Paul for the countless hacking queries, housing me and driving me through the winter to finish the thesis. To the patient and fantastic Andrea for all our collaboration meetings, both inside and outside of the pub. To Simon Poulding for his collaboration and for answering many stats questions. To the proof readers: Martin, Rob, Sevil, Ian, Peter, Jan and Iain (on so many occasions). Thanks to Paul Cairns for his tuition and advice on stats and experimental method. To Rob, Frantz and Paul for particular help in the summer when I needed it most.

To the EPSRC for their financial support, on grant EP/D050618/1. To everyone on the SEBASE project for their support and suggestions, including Per Kristian and Andrea: their dedication took them as far as Avalanche Peak. To Sean Luke and the ECJ team for producing the best evolutionary computation framework I could have hoped to find. To the support team at York for building a cluster to make my experimentation possible.

To my family for giving me the courage to do something so daunting, and for their unwavering support.

"You shall have joy, or you shall have power, said God;
you shall not have both."
Ralph Waldo Emerson.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

INTRODUCTION

1 MOTIVATION

1.1 LOW-RESOURCE SYSTEMS

Embedded systems are the dominant form of computing platform [Mesman et al., 2002]. In the past, the trend in embedded systems development has focused on miniaturisation [Heath, 1997]. However, for over a decade the focus has shifted towards other concerns, in particular the non-functional properties of theses systems. Ubiquitous consumer devices and technologies such as wireless sensor networks (WSNs) and radio frequency identification (RFID) chips are creating a new generation of very low-resource platforms where hardware capabilities are severely constrained [Sarrafzadeh et al., 2006]. Examples of such platforms are listed in Table 1. The reduced capabilities of these platforms result from very low production cost targets, physical footprint limitations, and the goal of maximising battery life by reducing power consumption.

Typically, the target markets for these devices are very demanding: they are produced in high volume, costs must be minimised, and time-to-market must be as short as possible. Products that share the same specification may contain different components. The same code or algorithm may be required to run on multiple different processors and embedded in different environments.

Application-specific requirements place further demands on the development of these systems. For example, it may be desirable or essential to minimise communication frequency, to meet hard and soft real-time constraints, or guarantee properties such as robustness, maintainability or security.

1.2 LOW-RESOURCE SOFTWARE

Software developers programming for these resource-limited systems are presented with the problem of ensuring functional correctness whilst simultaneously achieving the goals of satisfying multiple *non-functional requirements* such as those described above. This is made more difficult by the following concerns:

Device	Processor	Code Memory	RAM	Comms
Mica2 Wireless Mote	7.3MHz ARMega128L	128kB	4kB	38.4kbps
Crossbow Telosb Mote	8MHz TI-MSP430	16kB	10kB	UART, 250kbps RF
Xilinx Spartan-6 XC6SLX4 FPGA	n/a	3840 cells	216kb	n/a
Alien ALN-9640 RFID tag	n/a	n/a	800 bits	UHF

Table 1: Example low-resource platforms. See [Cro, Xil, Ali] for further details.

- There is a high-level of interdependency between requirements. For example, there may be a direct trade-off between memory and processor usage. Both memory usage and processor usage also affect power consumption, which may be reduced by varying processor speed [Jacome and Ramachandran, 2006].
- The relative importance of requirements may not be known. Approaching such problems by weighting the importance of each factor then creates the problem of how to decide upon those priorities and weightings [Berntsson Svensson et al., 2009].
- A single solution is vulnerable to change, as time-to-market is typically short and system requirements can shift rapidly [Villela et al., 2008].
- For a given problem, once a solution has been proposed it is usually not possible to determine if the suggested solution is optimal. Only by examining a set of alternative designs can the quality of a given solution be estimated [Henzinger and Sifakis, 2007].

1.3 TRADITIONAL LOW-RESOURCE SOFTWARE DEVELOPMENT

Whilst there has been a recent surge of interest in the design of both hardware and software for embedded systems, non-functional requirements have been addressed in the past with only a limited range of traditional methods. Techniques for satisfying all of the constraints and requirements of these systems are dispersed across multiple areas of expertise and are often applied separately to the design of different system components.

Traditional techniques usually focus on a single objective, or a single trade-off between two competing objectives. For example, a compiler may present configurations (such as GCC's "-Os" option) to prefer code size minimisation over time efficiency where those objectives conflict. The large number of potential trade-offs involved prohibits an individual engineer from considering the impact of a design decision on every requirement.

The point in the project lifecycle where these requirements are addressed also limits the impact that particular techniques or processes can have. Compiler-level optimisation, for example, is a post-implementation optimisation. As a result, the optimisations produced by a compiler tend to be localised, acting by transforming a small number of instructions rather than considering a more global view of the software. A compiler cannot alter decisions taken by the programmer that restrict what can now be achieved.

1.4 OPPORTUNITIES FOR IMPROVEMENT

Embedded systems software design has seen little focus on optimisation at the algorithmic level, where there are potentially large optimisation gains to be made from modifying or replacing a program or subprogram with a different form of solution. Rather, traditional methods have focused on small scale optimisation such as peephole optimisation found in modern compilers. This is because completely general and systematic methods for transforming software at a function-level (for example) do not exist.

In recent years, the field of program search, including its most successful branch of Genetic Programming (GP) [Koza, 1992], has shown some promise as a semi-automated method of program development. Integration of GP with multi-objective optimisation (MOO) techniques [Deb, 2001] allows the algorithm to be employed to explore trade-off surfaces, and the combination of MOO techniques with program search may prove a useful tool in engineering low-resource system software.

1.5 GENETIC PROGRAMMING

Program search is a heuristic search technique that attempts to automate the creation of program code to solve a problem, guided by a function that will measure the success of a particular program in solving the problem at hand. Genetic Programming [Koza, 1989] (GP) is the most popular program search algorithm, and since its conception in the late 1980s over 6000 papers have been published in the field [Langdon et al., 2009]. GP has been successfully applied to a range of application areas, and has achieved some impressive results [Koza et al., 2003].

Search algorithms such as GP can evaluate a larger number of potential solutions than human designers. They can also be used to solve multiple objectives simultaneously, and to find trade-offs between these objectives. Solutions can be presented "as is", or they can be utilised by human designers in gaining insight into a problem, for example by giving an estimate of the range of trade-offs possible. Hence search may accelerate the design process by focusing the engineer's effort on the most fruitful areas of the design space.

Previously applications of search to other problems within software engineering have been collectively known as Search-Based Software Engineering (SBSE) [Clark et al., 2003, Harman, 2007]. The most successful area of SBSE has been in the domain of software testing, and the techniques developed in that domain have shown potential in addressing non-functional properties [Afzal et al., 2009]. The success of search in these fields gives promise to the application of GP to non-functional requirements in the embedded domain.

Genetic Programming has previously been applied to novel embedded hardware design [Koza et al., 2005], and to non-functional requirements such as the size of source code [Langdon, 2000a]. However, its potential is far from fully exploited in this domain. For example, it has yet to be used to control dynamic properties of software such as power consumption.

1.6 RESEARCH GOALS

This research will investigate a new approach to designing software for low resource systems using multi-objective Genetic Programming. The techniques that this research aims to develop are intended to assist, rather than replace, the developer by providing possible solutions and information about trade-offs in order to help them gain insight into a problem.

1.7 STRUCTURE OF THIS DOCUMENT

This document is divided into four parts: this introduction, a literature review, experimental method and results, and a conclusion.

The final chapter in this part, Chapter 2, outlines the hypotheses of the thesis. These are the result of the literature review that follows, but they are outlined at the outset for clarity.

Part ii reviews the literature. After examining traditional methods of creating resource-efficient embedded systems, as well as recent and emerging techniques in Chapter 3, it proceeds to cover the two optimisation methods central to this thesis: Genetic Programming in Chapter 4 and multi-objective optimisation in Chapter 5.

Part iii contains three strands of experimentation designed to test the hypotheses in Chapter 2. The first, Chapter 6, attempts to create low-power software from scratch using Genetic Programming. The second, Chapter 7, aims to apply Genetic Programming to improve the execution time of existing software. Third and finally, Chapter 8 demonstrates the application of search in establishing fine-grained control over the non-functional behaviour of software.

Chapter 9 in Part iv reviews the hypotheses in the light of experimental results, and proposes future work to further develop and implement the techniques studied.

2 HYPOTHESES

2.1 INTRODUCTION

In this chapter, I detail the hypotheses that are the subject of investigation within the thesis. Firstly, I give an overall hypothesis, before subdividing this overarching statement into individual hypotheses that will be separately addressed.

The very nature of the hypotheses, and their relation to the solving of engineering tasks, means that this thesis must take the form of empirical investigation. In some cases the hypothesis cannot be conclusively proved or discredited: I can only present evidence that increases or decreases our confidence in its correctness.

2.2 THESIS HYPOTHESIS

The fundamental hypothesis of this research is:

> **Thesis Hypothesis**: *Program search is a versatile and effective method that can be used to satisfy the conflicting requirements of low-resource systems.*

In order to validate or falsify this hypothesis, I propose three separate strands of research:

1. Exploring the ability of GP to trade-off functionality against resource consumption for a given problem.
2. Determining whether GP can be used to improve existing software, in terms of its resource consumption.
3. Investigating the use of GP to gain fine-grained control over resource usage in general.

A hypothesis associated with each strand is now given.

2.2.1 *Capability of GP to make Trade-offs*

> **Hypothesis 1**: *Genetic Programming will be able to provide graceful degradation in the trade-off between resource consumption and functionality.*

"Graceful degradation" is not easily quantified, but in general we may consider that if we do not see a phase change in performance compared to resource consumption, or a resource threshold below which functionality will rapidly degrade, then GP is degrading gracefully. As an illuminating example, consider the problem of evolving sorting networks that will sort a list of inputs [Knuth, 1998]. If we restrict the number of possible comparisons below the known minimum required to completely sort the items, what can we achieve?

It is believed that GP will be able to achieve graceful degradation by combining variation of its current solutions with feedback from evaluating those individuals. It will then favour changes to an existing solution that give the most efficient returns in terms of providing functionality for a given resource level.

2.2.2 *Improving Existing Software using GP*

The potential of using Genetic Programming to improve existing software, rather than create new software, will be investigated. Such a line of research is interesting because it may prove more immediately applicable to deployed software and platforms. It is likely that this approach will require development of new search methods, or the novel combination of existing ones. The hypothesis of this research is:

> **Hypothesis 2**: *Genetic Programming will be able to optimise non-functional properties of software to a level not achievable by a compiler: in particular, solutions found by GP will Pareto-dominate hand-written solutions optimised by a compiler such as GCC.*[1]

For example, we may hope that GP can use handwritten C code as input, and produce optimised versions of that code consuming less power or taking a shorter time to execute. The rationale behind proposing such a hypothesis is that we may expect GP to manipulate code at a larger scale than a compiler, that it might recombine or replace subcomponents efficiently. In contrast, compilers are limited to a restrictive set of mappings from source to optimised assembly.

2.2.3 *Fine-Grained Control of Resource Usage*

Traditional optimisation methods attempt to reduce resource usage, that is they are concerned with transforming the resource usage of software or a system to one that is more resource-efficient. It may however, be desirable to exert much more precise control over resource consumption. For example, we may wish to produce a resource consumption pattern that relates the resources used to the numerical inputs of a program. This would have applications in computer security, resource-bound and resource-scalable functions.

The problem with obtaining such control is that precise resource consumption is an emergent property of software, the compiler, the system that executes it, and the machine state prior to that software's execution. Evolutionary computation has been show to cope well with such complex relationships in other domains, and therefore I propose Hypothesis 3:

> **Hypothesis 3**: *Genetic Programming can act as a mechanism to improve fine-grained control over emergent properties arising from the interaction between compiled source code and the host hardware platform by treating this system as a "black box", and discovering complex relationships through exploration of the search landscape.*

A human programmer would find such a task very challenging, because a modern processor and compiler constitute a complex system that makes it difficult to predict the impact of a change at source level

1 For the definition of Pareto-dominance, please see Chapter 5.

on the behaviour of the resulting compiled code. However, GP does not rely on logical analysis of a situation, rather it is a search mechanism that will attempt to find a gradient it can follow. A human equivalent to this would be a "trial and error" approach, but GP can automate this process and carry it out in parallel through its search operators. GP also does not carry the weight of experience or intuition, which may bias a designer's choices. Therefore, I expect GP to be effective in achieving this control.

2.3 SUMMARY

I have listed the hypotheses this research will answer. Evidence regarding Hypotheses 1, 2, and 3 will be presented in experimental chapters 6, 7 and 8 respectively.

Part ii of this thesis now reviews relevant literature in the area of traditional optimisation of non-functional properties, and the two search techniques central to the empirical work: Genetic Programming and multi-objective optimisation.

Part II

LITERATURE REVIEW

3 TRADITIONAL RESOURCE OPTIMISATION

3.1 INTRODUCTION

This chapter reviews previous approaches to creating resource-efficient embedded systems. Firstly, it covers the traditional methods used to optimise embedded hardware and software designs. Whilst not exhaustive, this provides an overview of the issues involved and the prevalence of trade-offs between competing concerns. Secondly, I examine areas where optimisation techniques such as heuristic search algorithms have previously been applied. Finally, I consider methods of evaluating designs and the limitations of current approaches.

3.2 TRADITIONAL METHODS

The approaches covered span a range of fields without a unifying methodology. Most techniques focus on a single part of the system, and a single non-functional property – typically execution time, memory or power consumption. The notable exceptions to such single-focus are *system synthesis* and in particular *hardware/software codesign* [Vahid and Givargis, 2008, Micheli et al., 1997], which is becoming increasingly important due to the indivisible nature of system optimisation.

Whilst the embedded system design literature [Ganssle, 1999, Heath, 1997] is principally concerned with *making decisions* that trade-off product costs, footprint size and technical capabilities, there is little emphasis on *actually exploring* trade-off spaces in both manual decision-making and automated techniques. Where automation is applied, the engineer is often not presented with a choice of trade-offs, rather they are given a single solution based on the approach a compiler author or hardware designer has taken. This amounts to a static selection of weightings in a multi-objective problem, as described in Chapter 5.

Increasingly, the importance of time-to-market has created a focus on reducing development time, achieved by using generic hardware, customising off-the-shelf components (for example, through programmable logic) and relying more and more on software to provide optimisations for a particular application. As software is the most expensive part of the system to develop, Ganssle [1999] suggests that systems may be built more effectively using a software-centric development process:

> "It's time to reverse the conventional design approach, and let the software drive the hardware design."

The most important non-functional properties in modern embedded systems design are in general power consumption [Jacome and Ramachandran, 2006], physical and memory footprint size [Clausen et al., 2000] and time efficiency, if we exclude "meta" properties such as production cost. Requirements of increasing importance are considerations of thermal properties, and security concerns regarding side-channel analysis [Voyiatzis et al., 2006]. Memory efficiency has become less of a concern as low-cost and physically compact RAM technologies have become available. There are exceptions, for example in the area of RFID design where the impact of memory requirements on cost is small but significant when producing large volumes of such heavily constrained systems.

The methods of providing resource efficiency are now examined in reverse order of their proximity to the application level. In this thesis, I target the application layer, but aim to transcend these divisions by simulating the effects of application changes on the system in a holistic sense. Low-level details such as transistor technologies are deliberately omitted, because they are usually beyond the control of the system designer.

3.3 HARDWARE DESIGN

3.3.1 *Processors*

Modern processors are differentiated by their Instruction Set Architectures (ISAs), and the level of integration they provide on-chip. The most striking design choice is between Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC) [Isen et al., 2009]. RISC chips dominate the market at present, although manufacturers such as Intel are introducing CISC implementations in small footprint, lower-power chip designs [Intel Atom Processor, 2009]. For the moment, RISC prevails and is able to produce smaller and usually more power-efficient processors, with satisfactory and increasing processing capability.

The uptake of RISC architectures was initially surprisingly slow. They were first proposed in the 1970s, based on systematic analysis of the code produced by compilers targeted for CISC platforms [Patterson and Ditzel, 1980]. One study [Alexander and Wortman, 1975] demonstrated that 80% of code used only 20% of the instructions provided, which became known as the 80-20 rule. CISC architectures had even grown to the extent that they included what Patterson described as *irrational instructions*, whereby a specialised instruction was sometimes less efficient than an equivalent sequence of simpler instructions within an ISA. However, RISC architectures were practically abandoned until a decade later. Fascinatingly, the reason for this was the limitations of the *compiler technology*, an observation that can perhaps be mirrored in the work presented in Chapter 7. Compilers were in part memory-constrained and also simply not sophisticated enough to produce efficient sequences of RISC instructions to perform the same functions implemented in a single CISC instruction. In a similar manner today, this thesis looks at one method that could be used to improve the optimisations a compiler can perform.

The choice of a RISC/CISC architecture presents a classic trade-off decision to the engineer. RISC architectures are generally more power-efficient and physically compact, but a larger number of instructions are required to implement the same function compared to a CISC architecture, and hence a greater amount of memory is used for code storage.

Within specific ISAs, there are often developments of subsets or extensions targeted for specific applications. The heavily promoted MMX technology [Peleg and Weiser, 1996] used in Intel's Pentium line of processors is a good example: extra instructions were added to targeted common functions used in handling multimedia. The ARM

Thumb instruction set extension is another example [Seal, 2000]. More generally, the concept of *instruction subsetting* can be used to implement fewer instructions, i.e. sacrificing programmability, in order to improve performance measures such as power consumption. Dougherty et al. [1998] examine just such a method, and look at a range of trade-offs rather than the two extremes. This is an example of hardware/software codesign, in that the instruction set is chosen partly on the basis of the software application.

Application-specific processors [Goodwin and Petkov, 2003] are becoming increasingly popular. The first examples of such tailoring of hardware to application were Digital Signal Processors (DSPs) arriving in the 1980s, specifically targeting applications that require high-throughput of data in order to perform filtering that was previously implemented in analogue circuitry. DSPs are far more efficient than employing a full microprocessor, and offer flexibility (another non-functional requirement) and low cost that analogue circuitry cannot.

Whilst ISA choice is a macro-level design decision, manufacturers have often resorted to much finer-grained optimisations. In the past 8-bit and (more briefly) 16-bit devices dominated the market. Manufacturing costs heavily dictated processor design, and processors such as the Motorola MC68HC05 went so far as to trim individual bits from physical register width to reduce the die size required.

A full history of processor design cannot be given here, but clearly the development of processor technology in general, such as pipelining and speculative branch execution, has impacted the field of embedded systems: most such improvements have been applied to lines such as ARM, PowerPC and Motorola's processors. One popular example is the voltage scaling used in PCs and embedded systems alike: see Section 3.4.1.

3.3.2 *Memory Subsystems*

Memory access is a major cause of latency in software execution, and accessing off-chip memory can involve a latency of anywhere from 2 to 10 cycles [Heath, 1997]. Memory access, along with any external communication, is usually the dominant cause of power consumption in an embedded system. This subsystem is also a key factor in determining footprint size. There are trade-offs contained within this system alone: for example, delays can be reduced by increasing the bandwidth of paths to memory, at the cost of increased power consumption. For a detailed review, see Benini et al. [2003].

Embedded systems usually provide a combination of read-only memory (typically an EPROM or cheaper One-Time-Programmables) for software storage along with working RAM. The latter is implemented using DRAM or SRAM technology, which provides another excellent example of the trade-offs involved: DRAM is much smaller (a single transistor can represent a single bit) in its footprint, but it must be constantly refreshed, resulting in higher power consumption and lower performance. This is because the refresh cycle must be synchronised with access to the memory.

Automated techniques can also be used to optimise a memory architecture by exploring the design space. For example, Coumeri and Thomas [2000] performed exhaustive search over a subset of the space of memory hierarchies for embedded SRAM. They used linear regression to construct power, size and performance models after evaluating the subset of possible designs using circuit simulators.

BUS DESIGN Bus design is critical in determining software performance, as memory access and bus contention can be a key bottleneck. Memory access, along with other external resource access, is the slowest activity in embedded systems. With the advent of processor pipelining, bus contention became very important and this was reflected in the separation of data and instruction buses in most systems.

The processor bus width can be increased to allow more data or instructions to be fetched in a single memory access and reduce execution time, or conversely it may be decreased to reduce the space required and the power consumption.

A serial bus may be employed to reduce the number of wires used, or the data transmitted may be compressed [Yoshida et al., 1997]. For example, Hatta et al. [2006] proposed replacing the instruction bus with a serial bus. This requires extra logic for encoding and decoding. They also employed differential data transfer, which uses a difference rather than absolute encoding of data to reduce the amount of information transmitted. They achieve impressive power savings of up to 66%, together with a reduction in footprint size over a conventional bus.

Yoshida et al. [1997] reduced system power consumption by compressing instructions using a table-lookup method. The frequency of each instruction in the software is analysed and a compression table constructed. Instructions are decompressed by a unit within the processor prior to execution. This reduces RAM access and results in a significant reduction in the amount of power consumed.

CACHE Caching instructions and data can be vital in improving performance by reducing stalls caused by external memory access. Off-chip memory access latencies are a major factor in performance, particularly as CPUs have grown in speed compared to memory technologies, and motivate the development of improved caching methods. Hardware prefetching retrieves data from memory based on current accesses. For example, as the end of the cached part of an array is reached it may request the next part of that array. This is augmented by *software prefetching* [Callahan et al., 1991], a technique that interestingly shifts optimisation of memory access away from hardware support and onto the compiler.

A design of increasing popularity in recent years is the inclusion of *scratchpad memories* [Panda et al., 1999]. The relative benefits of scratchpad memory over standard cache are explored by Banakar et al. [2002], and they evaluate the impact of the design decision on physical footprint and energy consumption. Scratchpad memory contents mirror parts of the main system RAM and are determined prior to execution by the programmer's design, that is a traditionally hardware-based mechanism is replaced by a compile-time optimisation.

Panda et al. [1999] explore the optimisation of the cache hierarchy for embedded systems based on static analysis of the software to be executed. They give pseudocode that examines the impact of decisions across three components: memory size, cache size and cache line size. Their method uses locality analysis and estimation of access costs to select the locations of main memory to be cached in scratchpad memory, and approximates the overhead of memory access given the structure of the cache hierarchy.

3.3.3 *Networks*

Embedded systems interact with networks via standard hardware components in a WSN or LAN, and they can also incorporate network-on-chip technologies [Kumar et al., 2002] that connect components of the embedded system itself. On-chip networks are necessary due to the increased number of components on a chip demanding high-speed interconnect within stringent wiring requirements.

Trade-offs can be made between communication and other properties of a system, in terms of the frequency and amount of communication required. Significant energy savings can be made; for example, network cards can be shut down, or placed in a low-power mode, and traffic shaping and compression can be used to reduce the quantity of data transmitted. Acquaviva et al. [2005] reduced the power consumption of a wearable computer by replacing standard 802.11b Wi-Fi wireless network power management with optimisation controlled by the server. As the activity of the client is predictable, the required quantity of data for a set time period can be transmitted in bursts, buffered, and the wireless card can then sleep, in an example of *collaborative power management*.

Shang et al. [2006] were the first to consider on-chip network optimisation in the context of thermal management. Their paper describes a simulator and run-time router management scheme that allows the design and control of on-chip routing mechanisms to prevent chip temperatures from reaching unsafe thresholds. The routing works by monitoring the temperature at points around the chip, and both throttling and re-routing data across alternative paths when temperatures are high at specific locations. There is a trade-off between temperature management and performance, as re-routing can increase network latency. The same technique could be applied to manage temperature more generally across all system components, in a similar manner to power management techniques.

3.3.4 *Peripherals and Interrupts*

Embedded systems, by their very definition, must usually interact with external devices such as serial/parallel interfaces and analogue to digital converters. These components can dictate the design choices of the system, as they produce streams of data that require processing or provision by system software. Mechanisms such as interrupts and Direct Memory Access (DMA) can reduce the load on the main processor by allowing external peripherals to access memory directly.

3.3.5 *Power Management*

A survey on power management techniques can be found in Benini et al. [2000]. Power management reduces power consumption by incorporating low-power modes in system components. For example, hard disks, memory components, displays and laptop processors have such modes, commonly controlled through the Advanced Configuration and

Power interface (ACPI). Components change between modes that offer different trade-offs between performance factors (such as operation speed, responsiveness) and power consumption.

For power management to be successful, workload demands on a system must vary over time and be to some extent predictable (often difficult to achieve) such that a policy may be designed to capitalise on expected idle time. Bouyssounouse and Sifakis [2005] state:

> *"Even though most of the publications on power management are concerned with design-time techniques, their usefulness in practice is quite limited. This is because in many practical cases, it is not possible to characterise the workload with the required level of precision."*

There is a gap between the creation of resource-efficient design methods and the reality of the way embedded systems are actually developed, which is ad hoc and most crucially usually involves fitting software to a predetermined hardware design.

Power management features were first introduced in mainstream processors in the 1980s, for example in the Intel 80386SL processor, which included a power control model [Heath, 1997]. The Motorola MPC603 is an example of one of the first embedded processors with power management. Designing power-manageable components has associated costs. Switching between different modes incurs transition costs, including extra power consumption in some cases, and transition times also affect overall responsiveness. It is possible to hide the delay of restoration from a power-saving mode if the end of the idle period is known in advance [Langen and Juurlink, 2007]

Zhang and Vahid [2002] give one example of a power-manageable component, a power-configurable bus. It enables a software-selected low-power mode that works by reducing the switching frequencies of both data and address buses. This sacrifices 10ns of fetch time but reduces the power consumption by up to 97% for a series of memory-intensive example applications. A reduction in clock speed is required, a small overhead compared to the power savings made.

3.4 OPERATING SYSTEM OPTIMISATION

Most embedded systems now contain an operating system, often a Real-Time Operating System (RTOS) that provides context switching, mutual exclusion and interprocess communication. Good OS design can be used to improve the efficiency of the system as a whole. For example, Park and Shin [2007] examine the impact of memory management configuration in Linux, in terms of its overhead on system calls and memory access, and demonstrate that reductions in execution time of over 20% on a set of benchmarks can be gained through reconfiguration.

3.4.1 *Scheduling*

Modern processors support voltage scaling, which allows the processor to run using different voltages at different clock frequencies. A lower voltage corresponds to a slower execution speed, with a subsequent power saving. Processors may support continuous variation of voltage, such as the XScale Series [2007], or discrete voltage settings. For example, the Intel 80200 can be varied from 1 to 1.5V in small increments, with a corresponding change in frequency from 200 to 700MHz [Jacome and Ramachandran, 2006]. Again, changing to more efficient configurations has a transition cost that must be taken into account.

Voltage scaling can be exploited in the design stage to statically minimise power consumption, or the scheduler can respond dynamically based on prior task execution data and heuristics. Jha [2001] surveys power-aware scheduling algorithms that exploit dynamic voltage scaling and dynamic power management.

Standard scheduling methods have been extended to incorporate energy trade-offs as well as time and value constraints. For example, Rusu et al. [2003] give a scheduling algorithm that is energy efficient, whilst Hung et al. [2005] incorporate thermal concerns into scheduling.

Yavatkar and Lakshman [1995] describe a Dynamic Soft Real-Time (DSRT) processor scheduler that is based on the supply of dynamic information from executing tasks in order to create a schedule. Applications provide an estimate of required execution time to a Quality of Service (QOS) manager, which cooperates with the scheduler to assign priorities and CPU allocation dynamically.

Increasingly, modern RTOSs integrate with hardware power management facilities (see Section 3.3.5) to provide a dynamic system-wide power management policy. Applications effectively provide real-time profiling information to the operating system; designing middleware to fulfil this task is currently a popular area of research.

Static power dissipation is increasing as chips incorporate more and more transistors when additional functionality is integrated. This is raising the importance of both dynamic voltage scheduling and power-saving CPU modes in conserving energy. Langen and Juurlink [2007] explore the trade-offs between the two power-saving methods. The most obvious technique of powering down for the slack after an earliest-deadline first schedule (known as "schedule and stretch") is *not* the most efficient.

3.5 compiler optimisation

Nearly all software optimisation within embedded systems is carried out by a compiler. Compilers optimise for processing speed, memory usage and, increasingly, power consumption. There is a balance to be struck between these factors, presented to the developer as a set of fixed trade-offs. Compilers are limited in their effectiveness by the information available about the software and by their place within the design lifecycle. The field of compiler construction and optimisation is vast: only a set of common optimisations is presented here.

In general, compilers optimise in two ways: firstly, by improving individual operations and sets of operations and secondly by laying out code and data in such a way as to reduce off-chip memory access. The former type of compiler optimisations are localised within a small section of the program, such as a sequence of consecutive instructions, removing unnecessary code and modifying other code. For example, unreachable code can be removed, or loops unrolled.

Compiler optimisations have significant implications for debugging: the code can be transformed to such a degree that it may no longer be possible to apply standard debugging tools. For example, instructions may be replaced with alternatives – such as using an OR instruction to zero a register. Symbol tables may be affected, making it difficult to translate between actions in assembler and the original high-level source code. It is this complex relationship between source and target machine code that makes it so difficult for an engineer to finely tune optimisations and trade-offs at the source-level. The difficulty developers have in estimating the size of compiled code given the original source illustrates the complexity of this relationship.

3.5.1 *Register Allocation*

Lowering power consumption and improving execution speed by replacing memory access with cache and register usage is one of the most effective types of compiler optimisation. Memory access usually incurs a large clock-cycle overhead, as well as consuming more energy through the use of external communication such as a bus interface to DRAM. By allocating data to registers and on-chip memory, as well as selecting register-based operations where possible, compilers can improve the speed and energy efficiency of a program.

Lee et al. [2005] improve allocation of data to registers by profiling program code, then producing traces of the most popular function calls. This reduces program execution time and program size. Reducing program size reduces the size of the ROM required in embedded systems and hence the cost and physical footprint of the memory.

3.5.2 *Cache Optimisation*

Compilers can optimise memory access by using cache to hold frequently accessed data. Data selection is based on *locality analysis*. For example, array access has a strong degree of locality as many algorithms will navigate the array by accessing successive locations in memory.

Unsal et al. [2003] introduce two techniques: the first is a combination of compiler-driven optimisations and scratchpad memories, while the second is a new cache architecture termed "cool-cache" that handles non-scalar (i.e. array) memory access. Static analysis of scalar values is used to utilise scratchpad memory efficiently, and both static and dynamic techniques are implemented to improve cache performance for non-scalar access. The techniques can optionally be supported by custom hardware design.

3.5.3 *Loop Optimisation*

Loop optimisation transforms loops within a program to improve efficiency. There are many types of loop transformations. One example is loop unrolling, whereby the contents of a loop with n iterations are repeated n times, removing the branch statement and conditional logic of the loop. This reduces the number of branches required, and the total number of executed instructions.

Another example is that nested inner and outer loops may be exchanged to improve the locality of data access. For example if a two-dimensional array is being explored, the underlying array representation will favour continuous access across one dimension over the other. This supports caching and improves the efficiency of memory access.

More examples of loop optimisation are described by Aho and Ullman [1977].

3.5.4 *Peep-hole Optimisation*

Peep-hole optimisation transforms a sequence of instructions based on a ruleset. The rules include substitution of alternative op-codes and rearranging existing instructions in order to improve efficiency. For example, a multiplication by a power of two can be replaced by a shift operation. More than one transformation may be applicable to the same set of instructions, and a different ordering of transformation applications can produce different results. Deciding the order to be applied is known as the "phase ordering" problem. Most compilers use a fixed ordering.

3.6 SOFTWARE DESIGN

Traditionally, hand-crafted (assembly-level) optimisations were made in order to improve the efficiency of code, but such manually intensive methods are now normally superceded by modern compiler optimisation. Short time-to-market requirements in the embedded systems industry have resulted in the adoption of high-level languages for implementation.

High-level software design can easily outweigh the impact of optimisations made at lower levels of the system architecture, and some examples of the impact modern software design can have at the high level of abstraction follow.

3.6.1 *Low-Energy Protocols*

Network protocols can be designed to be power and communication-efficient, as well as guaranteeing delivery and low latency. Wireless sensor network protocols in particular [Akyildiz et al., 2002] are the subject of intensive research, as communication dominates the power consumption of these very resource-limited platforms.

For example, Mathew et al. [2005] modify the bootstrapping part of a network protocol for Wireless Sensor Networks (WSNs) to reduce the amount of transmission required and reduce the time spent listening for incoming packets. They increase the power efficiency of the network by providing longer windows of idle activity, during which time sensor nodes can switch to low power modes.

In general, protocols for WSNs must aim to satisfy many objectives. As well as guaranteed delivery, they may try to minimise the number of hops taken to deliver a message from one node to another. This reduces the number of nodes that must wake up and consume power in order to deliver a message. These objectives may be met through designing network topologies that facilitate these objectives, such as clustering and routing graph planarisation [Zhao and Guibas, 2004].

3.6.2 *Energy-Scalable Algorithms*

Certain algorithms can be expressed in a scalable manner, to provide continuous trade-offs between non-functional properties and quality

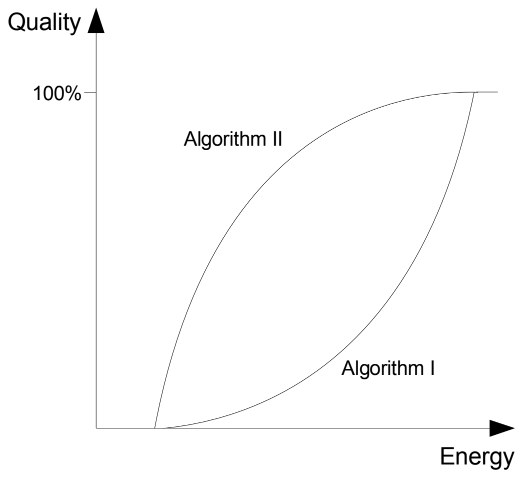


Figure 1: Visualising scalability in terms of energy versus quality.

of service attributes. Sinha et al. [2000] show how algorithmic transformation can be made to produce energy-efficient scalability in DSP applications such as filtering, image decompression and beamforming. Their notion of scalability requires incremental refinement, in the sense that an output is continually improved as energy is used, and an energy-scalable algorithm must be able to provide the largest returns at lower levels of energy, with returns diminishing as the energy consumption increases.

Figure 1, based on Sinha *et al.* , demonstrates how one algorithm may be more scalable than another. Algorithm II is more energy-scalable. They note the importance of such scalability:

> *"Algorithms that render incremental refinement of a certain quality metric such that the marginal returns from every additional unit of energy is diminishing are highly desirable in embedded applications."*

Resource-scalable algorithms are closely linked to the field of anytime algorithm research [Zilberstein, 1996], although anytime algorithms traditionally have focused on specific problems such as artificial intelligence applications.

3.6.3 *Resource-Efficient Design Patterns*

Noble and Weir [2001] provide a collection of design patterns for engineering low-memory software. These patterns are collected from engineers and developed through experience of engineering such systems. These higher-level modifications to system design can have a large impact on overall memory usage.

The techniques include the use of small memory architectures, taking advantage of cheaper secondary storage, compression, lightweight data structures and efficient memory allocation. The patterns are one example of explicitly defining aspects of the design process that have developed organically to satisfy non-functional requirements.

3.6.4 *High-Level Optimisation*

Much less work has been published on direct optimisation of program characteristics through altering source code, rather than the actions of the compiler. It is obvious that the choice of algorithm may dominate resource efficiency: this is the purpose of complexity theory, but there has been little work on improving efficiency at the high-level, at least in the context of embedded systems. One exception is the development of *portfolio* algorithms [Gomes and Selman, 1997], which partition the input space and choose an algorithm based on the input given, in order to minimise resource consumption (usually time) for that input.

3.7 full system synthesis

System synthesis is normally used to describe the process of selecting the correct hardware architecture and combination of components to construct an embedded system. This involves taking into account the software that will run on the platform, and other requirements such as cost, speed and energy consumption. Tools exist that partially automate this process, for example Carro et al. [2000] proposed a CAD environment that first analyses software to be run on a system and creates a profile that classifies a program as relatively memory, processor or data-intensive. Potential components are similarly characterised by their suitability for certain types of application. The CAD system can then suggest the most suitable application platform according to these profiles and the designer's requirements.

A more traditional and manually-intensive method of exploring the design space for a whole system is to use "suggest and simulate". A selection of proposed designs are supplied by engineers to be simulated using both mathematical and physical models. This technique has been used by NASA's Jet Propulsion Laboratory to engineer space systems, with demanding constraints on payload, trajectory, communication, mass, performance and risk. The results from such experiments give designers insight into the problem and feedback about their design choices. More recently, this work has begun to incorporate automatic optimisation techniques to reduce the length of the design process [Terrile et al., 2005].

3.8 limitations of traditional methods

Hardware techniques dominate current methods of ensuring resource-efficiency. These methods are in general limited in scope and usually assume a fixed set of tasks that will be executed in an embedded system. Whilst this was generally accepted in the past, modern embedded systems are increasingly based on general purpose hardware. For example, multi-processor system on chip (MPSoC) devices [Jerraya and Wolf, 2005] are designed for a range of media processing tasks and WSN motes must be flexible for use in different applications.

Established software optimisation is almost entirely focused on compiler-level optimisation, but as Aho and Ullman [1977] observed three decades ago:

> *"...the most important source of improvement in the running time of a program often lies beyond the reach of the compiler."*

This statement alludes to the improvements in efficiency that can be made at the algorithmic level, before the implementation is fixed and the compiler is required to optimise within the constraints of the implementation. Similarly, Kansal and Zhao [2008] present a tool to allow engineers to address power concerns at the algorithmic level of design, having noted the unexploited potential of application-specific power efficiency. These observations provide motivation for the research proposed in Chapter 2.

Compiler optimisation suffers from its position in the design process. Re-targeting compilers efficiently for new platforms such as application specific instruction processors (ASIPs) and DSP chips is not always possible due to time and budget constraints. Improving the ability of software to adapt to hardware platforms will therefore be of great benefit to these new platforms and their potential applications.

3.9 DESIGN EVALUATION

Once design decisions have been made, and system software has reached a functional state, the design can be evaluated. Typically, designs are evaluated using one or more of the following:

- Construction of a prototype system and inspection of that system.
- Emulation by substituting components with alternatives that can be controlled by a debugging system, such as an IDE on a PC.
- Full system simulation in software on a PC.

Inspection of a system may be performed unobtrusively by the use of probe measurement, logical analysers and oscilloscopes, or it may require modifying the system by instrumenting code and potentially substituting system components to facilitate testing. Using this method of emulation, it is often not possible to fully evaluate a design's performance as the timing properties of a system may be changed by manually stepping a processor or using an emulator. Very large amounts of data can be generated, and the difficulty of storing this onboard temporarily or feeding it to a host PC is another problem.

Emulation is difficult in practice, because substitute emulation parts may have different characteristics compared to the original components. For example, additional logic, leads and extended pins may cause extra delays, power consumption and timings may be slightly different, and voltages and impedances might not match the specification of the replaced component. This makes it difficult to debug subtle problems in the design, and also to accurately estimate non-functional properties such as execution time or power consumption.

Simulation of full systems has been limited in the past by available compute power. Although simulation remains computationally intensive, I am able to make use of large numbers of simulations when applying search methods in this thesis. For example, in 1995 one second of processing on a 25MHz RISC processor could have been expected to take around two hours of simulation time [Heath, 1997]. Nowadays it is possible to run simulations of x86 platforms at speeds that allow real-time use of Linux at a course-grained level of detail.

At the stage of evaluation, it is costly and time-consuming to redesign the hardware platform. Therefore, optimisation of the software for a particular platform must be performed beforehand using simulation, or else by accepting the hardware "as is" and adapting the software to it. The latter approach is more common, because it is usual for a hardware design and prototype to be delivered in advance of the software being written – and it is difficult or impossible to write working software without a hardware prototype with which to performing testing and debugging. More information on the realities of commercial embedded development can be found in Heath [1997] and Ganssle [1999].

3.10 OPTIMISATION USING SEARCH

In the last decade, researchers have begun to use optimisation methods such as heuristic search to aid the engineer in designing embedded systems, from hardware design to software implementation. Here I give representative examples of research in the areas previously discussed.

3.10.1 *Hardware Design*

Modern heuristic search and optimisation methods are being adopted by embedded systems researchers. Sheldon and Vahid [2009] use Pareto-based optimisation (see Chapter 5) to search the design space for FPGAs, recognising that such methods offer the opportunity for comparison between solutions and objectives.

3.10.2 *High-Level Software Design*

Risco-Martín et al. [2009] use evolutionary search to evolve dynamic memory managers for embedded systems. They effectively tailor a targeted system for the particular application software it will host, improving its memory access times, overall usage and energy consumption. This is in a similar vein to work by O'Neill and Ryan [1999], who evolved cache replacement policies using evolutionary search.

Li et al. [2005] investigated the use of Genetic Algorithms to evolve a hierarchical sorting algorithm that analyses the input to choose which sorting routine to use at each intermediate sorting step. This is effectively automatic creation of portfolio algorithms.

As work on traditional compiler optimisation has provided diminishing returns in efficiency, new techniques have target high-level transformations to improve source code. These techniques rely on stochastic selection of transformations, but only consider a very limited set of changes to the source code. For example, Franke et al. [2005] consider a subsequence of 81 possible transformations using a simple stochastic method to improve performance on digital signal processors. In Chapter 7, I allow arbitrary transformations to the source code and use more sophisticated search to provide a path to optimisations beyond more conservative approaches.

3.10.3 *Improving Compiler Performance*

There have been several attempts to apply the use of evolutionary techniques at the compiler interface, to find the most effective combination of compiler optimisations. Stephenson et al. [2003] used GP for solving hyperblock formation, register allocation and data prefetching. Leventhal et al. [2005] used evolutionary algorithms for offset assignment in digital signal processors, and Kri and Feeley [2004] used Genetic Algorithms for register allocation and instruction scheduling problems.

Compilers use sequences of code optimisation transformations, and these transformations are highly correlated to each other. In particular, the order in which they are applied can have a dramatic impact on the final outcome. The combination and order of selected transformations can be optimised using evolutionary algorithms: for example, the use of Genetic Algorithms to search for sequences that reduce code size has been studied by Cooper et al. [1999], Wild [2002], Kulkarni et al. [2004] and Fursin et al. [2008].

Compilers such as GCC give the user the choice of many optimisation parameters, and to simplify their choice, predefined subsets of possible optimisations (e.g., -Os, -O1, -O2 and -O3). However, the relative benefits of a particular set over another are dependent on the specific code undergoing optimisation. Hoste and Eeckhout [2008] therefore investigated the use of a multi-objective evolutionary algorithm to optimise parameter configurations for GCC.

3.10.4 *Evolvable Hardware*

Evolvable hardware is a relatively new discipline in electronics that first appeared in the late 1990s. Evolvable hardware uses evolutionary search in a similar manner to Genetic Programming described in Chapter 4. The seminal work in this area was performed by Thompson et al. [1999], whose initial work used a Genetic Algorithm to evolve configurations for FPGAs. They addressed both behavioural and non-behavioural requirements, the latter including size, weight, power consumption, construction cost, robustness (see also Harrison and Foster [2004]) and qualitative factors such as maintainability.

Thompson gives one of the reasons for adopting an evolutionary approach to hardware design:

> *"It is partly the ability to embrace non-behavioural requirements during all stages of an evolutionary design process, in combination with an exploration of new circuit structures and dynamics, that provides the opportunity for better circuits to arise through evolution."*

This is one of the motivations for adopting evolutionary techniques for software design in low resource systems. Furthermore, the division between hardware and software is blurring [Gorjiara et al., 2006], and as a result evolutionary hardware and software for domains such as embedded systems design are converging. In fact, Genetic Programming is a popular method of evolving hardware [Koza et al., 2004b, 2005], and so it is expected that the work presented in this thesis will also be applicable to programmable hardware development.

3.11 SUMMARY

In this chapter, I have summarised the traditional methods of optimising the non-functional properties of embedded systems. Such systems clearly occupy a large design space, composed of conflicting concerns and compromises. Much previous work has focused on optimisation of individual hardware subsystems, although co-design is becoming increasingly important.

Fine tuning of the impact of software on the non-functional properties of a system is often neglected, because it is manually intensive and resolving the large-scale issues of system design is difficult enough. As such, the focus is on high-level properties such as *the amount of communication* and thus power used by a node in a network, or the *scheduling demands* of software. These are important large-scale aspects of a system's non-functional behaviour, but the ability of software design to impact on this behaviour is yet to be fully exploited.

Individual instructions consume varying amounts of power, for example, and the interaction between software and the hardware executing it is a complex relationship that demonstrates emergent behaviour. Increased control over this behaviour may enable designers to improve the non-functional properties of systems, and to find trade-offs not previously locatable. How to achieve such a level of control? The next chapter introduces Genetic Programming as a means to this end.

4 GENETIC PROGRAMMING

4.1 INTRODUCTION

Genetic Programming refers to a set of stochastic heuristic search methods that can be used to locate expressions such as programs or mathematical functions. It does so by exploiting the information gained by repeatedly evaluating the quality of a population of potential solutions, and varying these solutions accordingly. It takes loose inspiration from Biology in its overall architecture and the operators employed to produce new candidate solutions.

It was Alan Turing [Turing, 1948] who in 1948 first proposed the use of a "genetical search" to automatically develop a machine by modifying its constituent functions, twenty years before Genetic Algorithms [Holland, 1975] were first used to tackle problems involving decision variables. Practitioners in Genetic Algorithms and the associated fields of Production and Classifier Systems went on to apply their search methods to program spaces, before more direct manipulation of program structures led to the advent of Genetic Programming.

Genetic Programming as a field arose in the 1980s, where papers by the likes of Smith [1980] and Cramer [1985] took the first steps towards a generalised method, but it was the voluminous work of Koza [Koza, 1992, 1995b, Koza et al., 1999, 2003] that crystallised the field, and Koza's terminology and many of his design decisions are still the *de facto* standard in modern applications.

In this chapter I first present an overview of the Genetic Programming method in its simplest, tree-based, form. I will examine the current theoretical understanding of the algorithm, before illustrating how it has been extended and modified by practitioners in an attempt to improve its performance, versatility, or the type of structures it evolves.

Further introductory material can be found in Banzhaf et al. [1998] and Poli et al. [2008].

4.2 AN OVERVIEW OF GP

GP could be described as a program search algorithm, a biologically-inspired optimisation method, or a problem solving technique that employs heuristic search. Turning away from the viewpoint of biological analogy, I present it here as a heuristic search method.

Michalewicz and Fogel [2000] insightfully explore the concepts of problem-solving using heuristic search, and reduce the essence of the techniques to the following components:

- The representation it uses for an individual solution.
- The method of evaluation used to rate each solution.
- The operations used to vary existing solutions and continue the search.

Whilst this is too limiting to encompass all potential search methods (consider hyper-heuristics [Burke et al., 2003], for example), it is certainly expressive enough to capture most existing algorithms.

Genetic Programming is a heuristic search method, and as such it consists of three central components: a representation, evaluation of solutions in that representation, and variation of those solutions. Due to its origins in biological analogy, a single cycle of evaluating and varying its current solution set is termed a *generation*. This is a useful term to describe a step in the search, although it does not necessarily correspond correctly to many implementations that rely upon *steady-state* evolution.

Note that Genetic Programming is an ambiguous label that can refer to a variety of algorithms, without any agreed ordering of effectiveness or relationship between problem characteristics and solution method. The strain most commonly encountered is undoubtedly traditional tree-based GP, and here this variety is presented before the alternatives.

4.3 representation

4.3.1 *Population*

GP maintains a set (population) P of candidate solutions. These candidate solutions are drawn from a search space X, which contains all valid individuals defined by the representation. $P \subseteq X$.

4.3.2 *Individuals*

An individual $p \in P$ is represented by a finite non-empty tree of arbitrary shape. The nodes represent functions, and the children of a node are inputs to their parent function. Leaf nodes are referred to as terminals, and can be regarded as functions of arity zero. In some of the literature a distinction is often made between non-terminal and terminal nodes. Here I make no such distinction and denote the set of possible functions as N.

As an example of a tree, consider Figure 2, which represents an expression to calculate the nth number in the Fibonacci sequence. It is equivalent to the following expression in Polish notation (with added parentheses for readability):

$$(\text{if } (\leq \; n \; 1) \; n \; (+ \; (\text{fib } (- \; n \; 1)) \; (\text{fib } (- \; n \; 2)))$$

Thus, in conventional tree-based GP, an individual is a rooted acyclic connected graph. The acyclic nature of the representation implies a lack of reuse: to use the same expression twice within a tree, it must be replicated in its entirety. This limitation is addressed by alternative representations and variation operators, described in Section 4.9.2.

Note that at this point, we have not expressed any constraints that refer to the type of arguments a function may require or return: that is, *closure of the function set* is a prerequisite such that all functions must be type-compatible with each other: usually this is achieved by ensuring all functions return and accept as arguments floating point values.

Functions can be arbitrarily defined, such that for example they may execute commands in a control system, sort data, or draw lines in a turtle drawing package [Koza et al., 2005, Agapitos and Lucas, 2006,

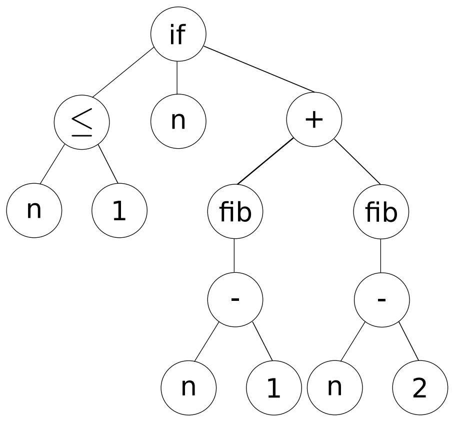


Figure 2: Tree representation of a function that calculates $\text{fib}(n)$.

Comisky et al., 2000]. One common inclusion is the use of Ephemeral Random Constants, or ERCs, which are used in a great deal of applications including symbolic regression. ERCs are constants initialised at random prior to execution of the main GP algorithm, based on the random seed supplied to the system. They therefore differ across runs with different seeds and may be combined, often ingeniously, by the search to produce other constant values.

In this simple representation, we have a single output returned from evaluating the symbolic expression tree. Multiple outputs are not catered for. One commonly implemented method of escaping this limitation is to rely on functions with *side-effects*, that is their evaluation affects the state of the executing program rather than just its final output, a step removed from a purely functional representation.

This representation is limiting when evolving programs with sequences of instructions. To implement sequences, it is common practice to introduce a node that may have several statements as its children, and in LISP this is achieved through the use of the PROGN function, which evaluates each of its child arguments in turn and then returns the result of evaluating its last argument. The potential use of these PROGN functions is mostly determined by the initial population. A fixed arity representation will not allow a new instruction to be added as an argument to an existing PROGN node and thus incremental changes to a series of instructions such as adding or removing a single instruction within a sequence cannot take place.

4.3.3 *Search Space*

Immediately we may see limitations in this representation: the same mathematical function can be represented by structurally distinct trees, thus there is much redundancy in the search space. The most simple example is the permutation of arguments to a commutative function. Unless our representation has a bias towards duplicating desirable functions more than undesirable ones, then it is an inefficient representation. This limitation underlies the issue of bloat (see Section 4.11) that is a major problem in practice.

The reason that this adopted representation is so inefficient is because its origins are in programming languages such as Lisp, and hence a human-readable representation was adopted rather than one that creates a minimal search space. This representation stands in stark contrast to the Genetic Algorithm [Holland, 1975], GP's close cousin and a very compact representation based on a vector of decision variables.

This representation also places restrictions on the types of functions we may use. For example, we cannot use conventional division due to the closure requirement, or else we may divide by zero. Therefore practitioners usually implement a form of protected division [Koza, 1992]. In work described in Chapter 7, we encounter similar *evaluative* or *run-time errors* that cannot be prevented, and instead individuals must be punished by modifying their fitness.

Another feature is the discrete nature of the representation: an instruction is either present or it is not; it is located prior to or after another instruction. In particular, the heavy dependence on the ordering of instructions and their interactions can make the search discontinuous. Fogel and Atmar [1990] observed the importance of continuity:

> *"Successful adaptive procedures must retain a sufficient link between parent and offspring to ensure that advances are maintained."*

The operators described in Section 4.6.2 must provide *meaningful* variation. The parents and offspring should be semantically similar, and the operators must be capable of exploring the search space in a continuous manner. This choice of representation creates a challenging problem of selecting a method of variation.

SEARCH SPACE SIZE The size of the search space considered is limited by the maximum depth of a tree, or else by the maximum number of nodes it can contain. Taking the former approach, we can specify $c(d)$, the number of possible trees of depth d, by the following recurrence relation:

$$c(d) \begin{cases} n_0 & \text{for } d = 1 \\ \sum_{a=0}^{max} n_a \cdot c(d-1)^a & \text{for } d > 1 \end{cases} \tag{4.1}$$

Here, n_a is the number of functions in N that have arity a. max is the maximum arity of functions in the function set. This formula is a generalisation of one for binary trees found in Ebner [1999], and can also be found in a tutorial given at the GECCO 2009 conference by Poli and Langdon [2009].[1]

We may consider some typical search space sizes. For an experiment using Boolean functions and including `True`, `False`, `AND`, `OR` in the function set, the number of possible trees is 2, 10, 202, 81610 and 3.5×10^{20} for tree depths of 1 through 5 respectively.

Consider the size of the largest search space for the experimentation reported in Chapter 6. There are over 5.07×10^{125421} possible trees in the search space, given a maximum depth of 17 (actually, this is

1 Acknowledgments to Jeremy Jacob for a simplification of this formula.

a conservative estimate – assuming only a single instantiation of an ephemeral random constant). The search space for GP is vast.

This equation also tells us that functions with a high arity have a strong impact on the size of the search space. This may lead us to conclude that it may be preferable to avoid high-arity functions when designing our function set. It also illustrates the importance of depth in determining search space size, which is the reason that it is used as a limiting parameter in standard toolkits. The maximum tree depth dominates the memory requirements of a GP system.

Langdon and Poli [2002] have demonstrated both empirical and theoretical results that examine the distribution of program behaviour as a function of increasing program length. Program behaviour converges on a fixed distribution as the size grows, at which point increasing the maximum program length does not alter the density of solutions within the search space. This ratio may be critical in determining the difficulty of a task: benchmarks such as the parity problem have *needle-in-a-haystack* search spaces, whereas other functions have higher solution densities. GP will not be able to outperform random search in any situation unless there is a guiding gradient, such as the existence of building blocks within the search space, or a fitness "hill" that can be climbed through crossover and mutation.

There is an unstated assumption we may make when designing our function set: we assume that solutions exist within the search space. In some applications of Genetic Programming, such as those found in Chapters 6 and 7 in this thesis, we may be aware of an existing solution within the search space. In other cases, such as in Chapter 8, we cannot be so sure.

4.4 INITIALISATION

The initial population P_0 must be generated prior to the commencement of the main algorithm. This is most often achieved with Koza's *ramped half-and-half* method [Koza, 1992], which itself is a composite of two individual methods: the *grow* method and the *full* method. Both methods rely on a maximum depth parameter to be chosen, which is varied when employing Koza's ramped half-and-half method.

The full method generates full trees such that all leaves are at the same depth, by selecting non-zero-arity functions from the function set at random when generating the tree and restricting that choice to zero-arity functions at the final depth. Pseudocode for the full method is given in Algorithm 1.

The grow method works similarly, but does not generate full trees. Instead, the choice of shape of tree is determined stochastically by the selection of functions from N. If a terminal is chosen, then that particular branch of the tree can be "grown" no further. If the function set contains a high proportion of zero-arity functions, then shorter trees will dominate. Pseudocode is given in Algorithm 2.

Koza's ramped half-and-half method is given in Algorithm 3. Here, the maximum depth for a particular tree is chosen at random up to the depth specified, and then one of the two methods is selected randomly according to a parameterised probability.

The heavy reliance on stochastic initialisation may well be questioned, as it does not incorporate any prior knowledge about the problem at hand (see Section 4.4.1), and it will also introduce a bias towards

Algorithm 1 Full Tree Initialisation.

```
1: function FULL($maxdepth$)
2:     if $maxdepth = 1$ then
3:         $node \Leftarrow n \in N,\ arity(n) = 0$
4:     else
5:         $node \Leftarrow n \in N,\ arity(n) \neq 0$
6:         for $i = 1$ to $arity(n)$ do
7:             $addChild(node, full(maxdepth - 1))$
8:         end for
9:     end if
10:    return node
11: end function
```

Algorithm 2 Grow Tree Initialisation.

```
1: function GROW($maxdepth$)
2:     if $maxdepth = 1$ then
3:         $node \Leftarrow n \in N,\ arity(n) = 0$
4:     else
5:         $node \Leftarrow n \in N$
6:         for $i = 1$ to $arity(n)$ do
7:             $addChild(node, grow(maxdepth - 1))$
8:         end for
9:     end if
10:    return node
11: end function
```

Algorithm 3 Ramped Half-and-Half Initialisation.

```
1: function RAMPED($maxdepth, growprob$)
2:     $depth \Leftarrow randint(1, maxdepth)$
3:     if $rand(0,1) < growprob$ then
4:         return $grow(depth)$
5:     else
6:         return $full(depth)$
7:     end if
8: end function
```

smaller trees over the depth range as the grow method's depth will be dependent on a series of Bernoulli trials. The latter bias can be removed by using an improved method such as Langdon's ramped uniform initialisation [Langdon, 2000b], at the cost of extra computation.

4.4.1 *Seeding*

Alternative methods of seeding exist, based on solutions created manually and also through other automated methods. The results of previous heuristic search algorithms such as Genetic Algorithms [Langdon, 1996a,b] and depth first search [Westerberg and Levine, 2001] have been used, and in both cases superior results to randomised initialisation were found. Manually created solutions were used as a starting point in Langdon and Nordin [2000] and Marek et al. [2003].

In Chapter 7 I suggest and test some alternative seeding methods that take an existing manually written solution and manipulate it to create the initial population. Subsequently, Schmidt and Lipson [2009] have also examined some of these methods.

It is well known that the starting point of a search within the solution space can have a large impact on its outcome. It may be considered surprising, then, that more research has not focused on the best methods to sample the search space when creating the initial generation within GP. The crucial importance of domain-specific knowledge in solving optimisation problems is also clear; yet still it is the case that in general little sound advice can be given on how best to incorporate existing information, such as low-quality or partially complete solutions to a problem, into an evolutionary run. There are examples in the literature where such seeding has been employed [Koza et al., 2004a].

The most relevant application of seeding in the literature is from Langdon and Nordin [2000], who employed a seeding strategy in order to improve one aspect of a solution's functional behaviour: its ability to generalise. Their initial population was created based on perfect individuals, where the goal of the evolutionary run was to produce solutions that were more parsimonious and had an improved ability to generalise. This mirrors the concern with improving a separate non-functional aspect of existing software in Chapter 7.

4.5 EVALUATION

The fitness of an individual program, $f(p)$, is a numeric representation of its ability to solve the problem at hand and is usually evaluated at each generation. This is achieved through the use of a fitness function, also known as an evaluation function or cost function (when it is to be minimised). To evaluate the function, an individual must be interpreted or executed, and the fitness assigned as a result. A compilation stage may be required, and the fitness evaluation of a program in general is more computationally expensive than in other heuristic search algorithms. We may also include some kind of state, such as an array or temporary data. This increases the memory and computational requirements of evaluation.

It is often the case in GP that an individual will be assigned a fitness derived from the results of executing a range of test cases. For example, a common application is symbolic regression, where an individual will be evaluated on a set of datapoints, and the squared error between the observed and desired output is used as a fitness value to be minimised. This is important, because it has implications in regard to the No Free Lunch theorem (see Section 4.14).

The choice of whether a high or low fitness value is considered desirable is arbitrary, though it would appear to be sensible in most cases to use a cost function, as the overwhelming majority of GP applications measure fitness as a reduction in error or resource consumption. It is usually possible to define the lowest possible error value, whereas determining an upper cost value for an individual in a way that may generalise to other problems is not normally possible.

It is standard procedure in the field to normalise fitness values, so that they lie in $[0, 1]$. Koza [1992] defines *raw fitness* as the initial score, perhaps the squared difference between the expected and actual values across a range of data points, for example. *Standardised fitness* restates the raw fitness if necessary as a cost function. *Adjusted fitness* reverses this and furthermore normalises the fitness as such:

$$a(p) = \frac{1}{1 + s(p)} \tag{4.2}$$

Here, a is adjusted fitness and s is standardised fitness. Koza refers to *normalised* fitness, yet another measure, as the fitness of an individual over the summation of the fitnesses of all individuals in the population.

The drawback to such normalisation is that the figures become unintuitive to the practitioner. With large fitness values, there is the risk of over-stretching the resolution of floating point representations. When combining multiple objectives or fitness values (see multi-objective optimisation in Chapter 5), there is also a danger that we may lose sight of the problem of setting relative weightings, and the fact that those weightings exist implicitly even if we use normalisation of some kind.

4.5.1 *Fitness Landscapes*

In introductory search texts, it is common to find *fitness landscapes* used as an aid to understand the behaviour of an algorithm. This is typically illustrated using a 3D plot with two axes referring to two decision variables and the remaining axis giving the fitness of an individual, a measure of its quality. An example is shown in Figure 3. If we consider an algorithm that works locally, then we can visualise its behaviour as selecting decision variables and evaluating the fitness function at each point. In this way, an algorithm may "climb" up a fitness gradient by repeatedly selecting adjacent points in the (x, y) plane and following the improvement in fitness value.

However, this analogy is not useful in explaining the behaviour of Genetic Programming. Firstly, because the dimensionality of the search space is vast: it is equal to the maximum number of nodes a tree may contain. Consider a binary tree of depth 20: the equivalent plot would have 1048576 dimensions. Second, GP is a population-based or *global* search algorithm, where there is no single step from one solution to the next without involving the other individuals in the population. As such, it is not possible to order trees on such a plot in a way that creates a meaningful geometric proximity between points within the search space. The potential neighbourhood of subsequent points to sample is in fact *the entire search space* if arbitrary mutation is permitted (see Section 4.6.2).

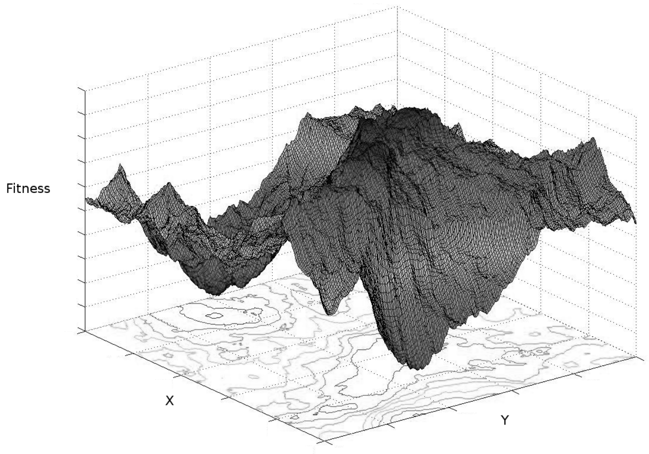


Figure 3: Example Fitness Landscape, from the Huygens Search and Optimisation Benchmarking Suite [2007].

4.6 variation

To create new individuals, that is to explore the search space, Genetic Programming provides a relation that stochastically maps one population onto another, to create the next generation of individuals. There is a sense of *neighbourhood*, as we might find in other search algorithms such as evolutionary strategies or simulated annealing. However, the neighbourhood size is potentially enormous, due to the representation we saw in Section 4.3. Until a better understanding of GP's crossover operator is developed, we can only regard the variation operators as mutation procedures that search an area loosely determined by an individual's existing tree structure [Banzhaf et al., 1998].

To carry out variation in an evolutionary algorithm, we must first select those individuals that we wish to use as the basis for the generation of new potential solutions, and we must then decide upon how to change those individuals to create new solutions.

4.6.1 *Selection*

Selection is invariably based on fitness, following a traditional Darwinian viewpoint of survival of the fittest. The most common operator in Genetic Programming publications is tournament selection, probably due to its presence in several major framework implementations, as well as its simplicity and the fact that it may be implemented in a distribution fashion without knowledge of state of the entire population.

tournament selection Tournament Selection selects a group of n individuals uniformly from the population with replacement such that the same individual may be selected more than once. The fitnesses of the individuals are then compared, and the best individual is selected from the group. We may vary n to increase the *selection pressure* on the population.

In experimentation reported in this thesis I did not find that a large tournament selection size led to any significant improvement in performance (see Chapter 7).

ROULETTE-WHEEL SELECTION In roulette wheel selection [Goldberg, 1989], individuals are chosen from the population in proportion to their fitness as follows:

$$prob(selected(p)) = \frac{f(p)}{\sum_{k=1}^{|P|} f(p_i)} \tag{4.3}$$

This method is simple to implement, but it may lead to a single individual dominating the population depending on the absolute fitness values used to carry out the selection. Rank selection aims to improve upon this method.

RANK SELECTION In rank selection, the probability of an individual being selected is based on its fitness ranking within the population, rather than their absolute fitness value. The individuals are ranked by fitness, 1 being the lowest rank, and then a mapping from this ranking to their probability of selection is applied. Ties between individuals with the same fitness values may be allowed for:

$$f'(p) = \frac{r_{max} - r_{min}}{|P|} \cdot r_i \tag{4.4}$$

The ranking is interpreted in many ways, for example the following exponential function is taken from Eiben and Smith [2003] given a constant scaling factor k:

$$prob(selected(p_i)) = \frac{1 - e^{r_i}}{k} \tag{4.5}$$

4.6.2 *Genetic Operators*

To explore the search space, new individuals are created based on those individuals selected as described in the previous section. It is important to recognise that it is a *copy* of the individuals that is passed to the operators: selection is performed with replacement, and duplicate individuals may be created by repeated application of the operators.

In GP the following three operators predominate, whilst alternatives and variants proposed have not been widely adopted:

- Crossover,
- Reproduction, and
- Mutation.

Usually, crossover between two selected parents is used to create two children, with a given probability, $prob(crossover)$. Reproduction is used instead with probability $1 - prob(crossover)$. Subsequently, mutation is applied to each child with probability $prob(mutation)$: importantly

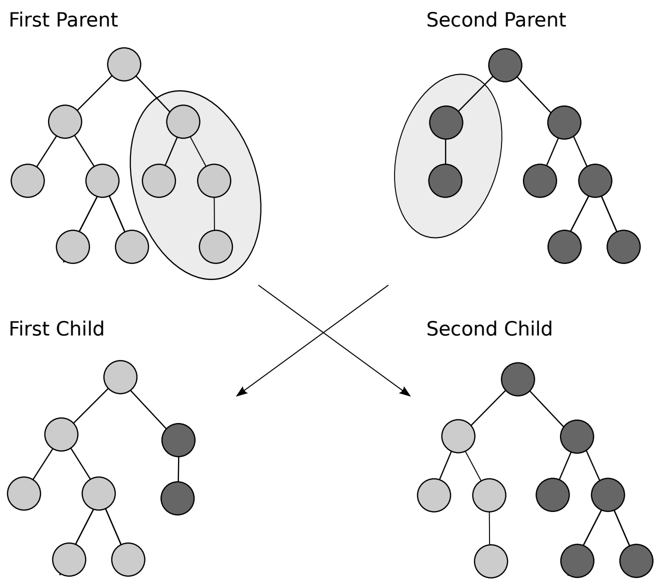


Figure 4: Crossover in Genetic Programming.

this is on a *per-individual* basis rather than a *per-node* basis, differing significantly from Genetic Algorithms.

Crossover is illustrated in Figure 4. A node is independently selected within each tree, and the two subtrees rooted at that node are exchanged. Reproduction passes a copy of each parent into the next generation without alteration. Mutation selects an individual node with a tree and replaces it with a randomly generated subtree as illustrated in Figure 5. The new subtree is usually created through the methods used in standard initialisation algorithms (see Section 4.4).

Both crossover and mutation can be constrained according to the probabilities of selecting a particular node. For this purpose we may differentiate between the root node, internal nodes or leaf nodes, with an associated probability of choosing each.

The role of crossover and mutation in exploring the search space is controversial. Banzhaf et al. [1998] refer to crossover as "the eye of the storm", as it is not clear that it is anything more than a macromutation operator; however for some problems I have empirically demonstrated that a crossover-based search outperforms a mutation-based algorithm [White and Poulding, 2009].

4.7 GP ALGORITHM SUMMARY

Given that the components of the algorithm have been introduced, pseudocode for Genetic Programming is given in Algorithm 4. The practitioner must choose *termination criteria*, that is a Boolean function that determines if the search may halt. This function may simply always return false, if it is desired that termination should occur after the maximum number of generations has been executed. Alternatively, the search may terminate if an adequate (or theoretically optimal) fitness value has been reached. For example, in a regression problem we may

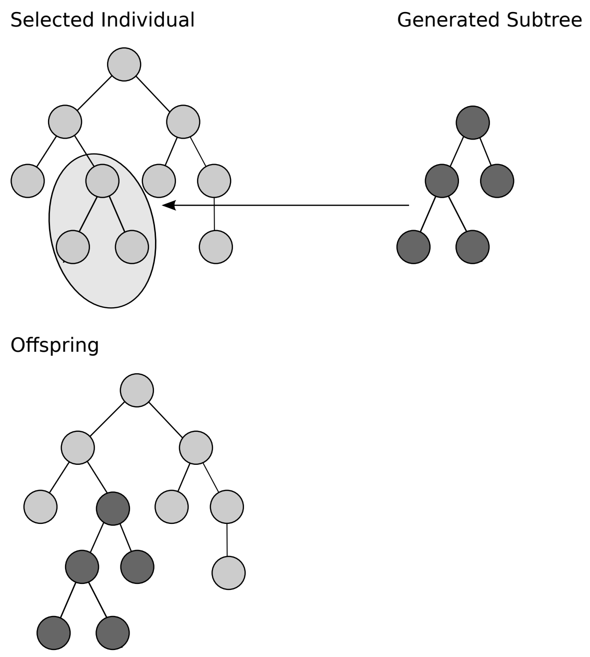


Figure 5: Mutation in Genetic Programming.

seek to find a system that matches the observations (utilised as test cases) within a desired precision.

4.8 THEORETICAL BASIS

Having described the basic algorithm, it is not immediately clear as to *why* we should expect Genetic Programming to find good solutions. The lack of a satisfactory answer to this question is also the root cause of the plethora of GP variants available and the lack of consensus as to which proposed extensions to adopt and those to discard.

The success of the algorithm is inevitably dependent on the skills of the practitioner: for example, the search space defined by a choice of function set must contain desirable solutions. However, given sufficient care in the construction of the function set, why should we expect GP to outperform random search, for instance?

This question is not yet fully answered, although significant advances in Genetic Programming theory have been achieved in the last few years. There are two main lines of attacking this question in the literature:

- Schema theories.
- Markov Models.

In addition, empirical analysis of specific search spaces provides useful insights such as Langdon and Poli's observations on the limiting distribution of program behaviours as their size increases, which they later formalise [Langdon, 2002, Langdon and Poli, 2005].

Algorithm 4 Pseudocode for Genetic Programming.

```
1: $t_{max} \in \mathbb{N}$
2: $\{p_c, p_m\} \in [0,1]$
3: $t \Leftarrow 1$
4: initialise population $P_1$
5: **for all** $p_i \in P_1$ **do**
6:     evaluate fitness $f(p_i)$
7: **end for**
8: **while** $t \leq t_{max}$ and termination criteria not satisfied **do**
9:     **while** $|P_{t+1}| < |P_t|$ **do**
10:        select parents $p_1, p_2 \in P_t$ by some fitness-based selection
11:        **if** $rand \leq p_c$ **then**                    $\triangleright$ $0 \leq rand \leq 1$
12:            child $c_1 \Leftarrow p_1$ xo $p_2$
13:            child $c_2 \Leftarrow p_2$ xo $p_1$
14:        **else**
15:            children $c_1, c_2 \Leftarrow p_1, p_2$
16:        **end if**
17:        $P_{t+1} \Leftarrow P_{t+1} \cup \{c_1, c_2\}$
18:    **end while**
19:    **for all** $p_i \in P_{t+1}$ **do**
20:        **if** $rand \leq p_m$ **then**
21:            mutate $p_i$
22:        **end if**
23:    **end for**
24:    **for all** $p_i \in P_{t+1}$ **do**
25:        evaluate $f(p_i)$
26:    **end for**
27:    $t \Leftarrow t + 1$
28: **end while**
29: **return** $\{p_i \in P \mid \forall p_j \in P, f(p_i) \leq f(p_j)\}$
```

4.8.1 *GP Schema Theory*

Schema theory is inspired by similar analysis in the field of Genetic Algorithms [Holland, 1975]. It is an attempt to describe the operation of the search algorithm in terms of the exploration of subsets of the search space defined by tree shapes and some of their constituent nodes. By relating the concentration of certain forms of tree from one generation to the next, schema theorems hope to demonstrate that the algorithm will on average explore trees of improving fitness. The canonical form of the Genetic Algorithm schema theory is given by Goldberg [1989]:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H) \cdot p_m \right] \tag{4.6}$$

This inequality relates the current concentration of a schema at generation t to its subsequent concentration at generation $t+1$. A schema in a binary string GA is a specification of some values of the digits in the string, so that any solution containing those specified digits belongs to that schema. The inequality states that the number of individuals m sampling the schema H at the next generation $t+1$ is proportional to the relative fitness of individuals within the schema and two other terms. Those terms represent the probability that the schema will be disrupted or "broken up" by crossover (dependent on the defining length δ of a schema, i.e. the probability that a crossover operation will separate the specified values) and the probability mutation will move an individual outside of a schema (i.e. the number of points with specified values, termed "order" and denoted o).

The related *building block hypothesis* states that building blocks (substrings in a binary GA) that perform well in different individuals will be exchanged amongst the population and combined with others to produce fitter individuals.

The difficulty with applying schema theory to GP is the problem of defining a schema in a tree representation, and justifying the idea that subtree crossover will meaningfully explore these schemata. A definition must also incorporate the notions of "defining length" and "order" to assess the probability that it will be disturbed by variation operators.

Different definitions of schemata for Genetic Programming are proposed by O'Reilly and Oppacher [1994], Rosca [1997], Whigham [1995] and Poli and Langdon [1997]. The approaches generally fall into two categories: those that consider schemata as individuals containing the same subtrees, and those considering schemata as rooted subtrees with "don't care" element placeholders. The latter is far more restrictive on the membership of a particular schema, but results in a more tractable problem. A thorough discussion of early GP schema theory is given in Langdon and Poli [2002].

SUBTREE SCHEMATA Koza [1992] suggests a GP equivalent of GA schemata:

> *"...a schema is a set of LISP S-Expressions containing common features..."*

In defining the schema, Koza does not use wildcards, as is the case in GA schema. By Koza's definition a subtree is rooted from one point

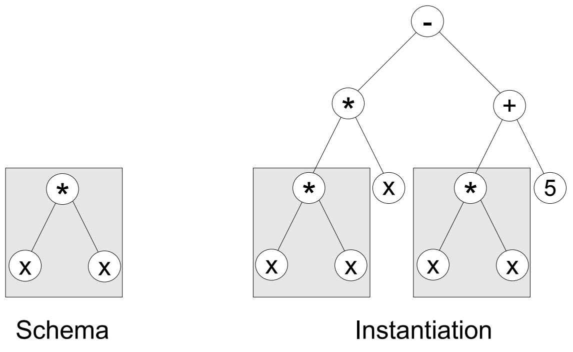


Figure 6: Koza's schemata as subtree building blocks.

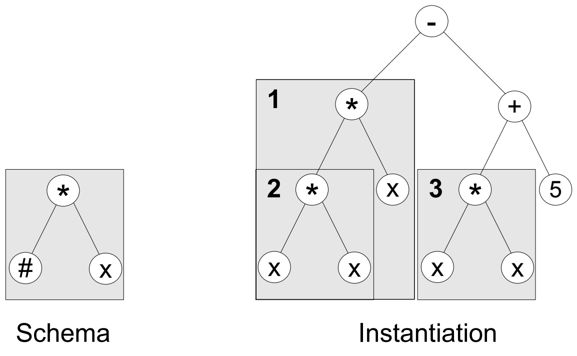


Figure 7: Schemata as tree fragments.

within the containing tree and extends to the terminal nodes. An example of a Koza schema and its instantiation within an individual is given in Figure 6. The schema is instantiated in two places within the individual. These multiple instantiations differ greatly from the traditional schema definition of Genetic Algorithms.

Contrastingly, O'Reilly and Oppacher [1994] introduce the concept of *tree fragments* as potentially incomplete subtrees. As in Genetic Algorithms, a schema may be defined using wildcards, where a "#" symbol represents a "don't care" position. For example, (+ # b) includes subtrees such as (+ 1 b) and (+ (* a 2) b). A "#" can be replaced by any subtree. Extending this concept, O'Reilly proposed that a schema be an ordered list of multiple fragments, all of which must be instantiated within an S-Expression in order for that expression to be considered a member of the schema. A schema may also require that a particular subtree exists in a specified number of places within a containing tree.

An example of a tree-fragment schema is given in Figure 7. The individual instantiates the schema three times. If a schema strictly specifies that fragments must occur a number of times, such as {(*(# x)),2}, then the situation is slightly different and illustrated in Figure 8. Note that in both cases the schema is instantiated in three different ways, and such cases highlight the difficulty of analysing such schemata.

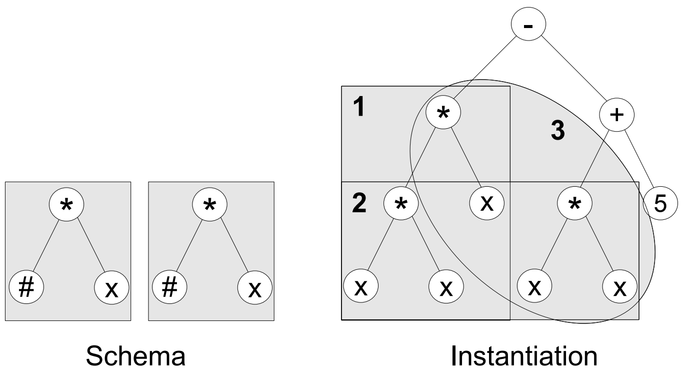


Figure 8: Schema composed of multiple tree fragments.

It is apparent that the defining length and order of a schema will vary between instantiations in O'Reilly's interpretation. This means that the same conclusions derived from the GA theorem cannot be drawn because the impact of this factor cannot be estimated, despite O'Reilly's formulation of a schema theorem. Perhaps more importantly, O'Reilly notes that:

> *"...it is empirically questionable whether building blocks always exist because partial solutions of consistently above average fitness and resilience to disruption are not assured..."*

4.8.2 *Rooted Tree Schemata*

An alternative definition (and perhaps the most successful) of schemata is of rooted trees such as that proposed in Poli and Langdon [1997] alongside initial work by Rosca [1997]. This definition is more closely related to the schema theory of GAs. Rooted schemata are complete trees composed of function nodes, terminal nodes, and don't-care symbols conventionally (and somewhat confusingly) marked by "=" . The don't-care symbol can be replaced only by a single node, not an arbitrary size subtree as in O'Reilly's definition. Therefore, an instantiation of the schema will be an individual tree with the same size and shape as the schema itself. Figure 9 illustrates one rooted schema and its instantiation. The schema has order 8 and a defining length of 11. The order of a schema is the number of nodes that are not marked as "don't care" functions, and the defining length of a schema is the total number of nodes in the schema.

Originally, Poli introduced the concept of one-point crossover for GP in order to restrict the disruption that crossover could have on schema membership. A similar restriction was used by Whigham [1995] in earlier work. Later, in Langdon and Poli [2002] this restriction was lifted and the potential for construction of schema instantiation is taken into account.

Through the introduction of a new Cartesian model of schemata, Poli and McPhee [Poli and McPhee, 2003] produced the first schema theorem for standard subtree-exchanging crossover and used the results to provide individual insights into features of Genetic Programming. This work is currently the most complete schema theorem for Genetic Pro-

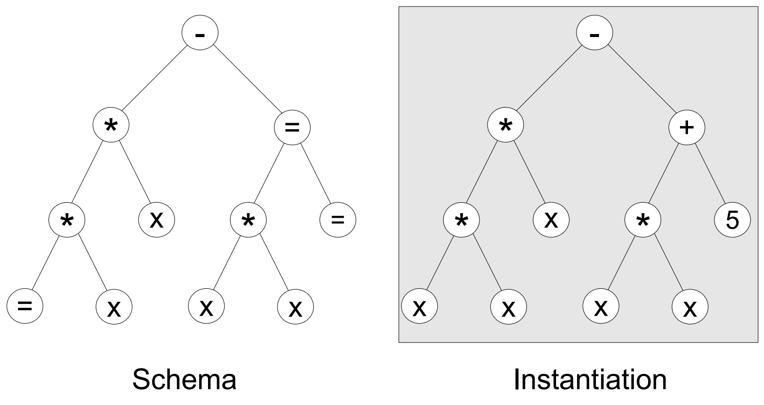


Figure 9: Rooted schema.

gramming, although it is still open to some of the traditional criticisms of schema theory.

4.8.3 *Limitations of GP Schema*

The problems with attempting to devise a schema theorem for Genetic Programming lie in the fundamental differences between Genetic Algorithms and Genetic Programming.

When using GAs to find an optimal combination of parameters encoded as a binary string, provided one-point crossover is used, the exchange of genetic material will always lead to a straightforward exchange of values at given positions on the genome. Contrastingly, this will not be the case with subtree building blocks in GP and it is not necessarily valid to assume that a subtree that contributes well to overall fitness in one position of an individual will similarly contribute well to fitness in another position within a similar individual. The rate of dispersal of good building blocks through the population will therefore rely on the ability of those building blocks to "perform" in any program position they find themselves within. As a procedural program is by its definition linear in operation, we cannot assume that most segments of sequential program code have this desirable property.

4.8.4 *Markov Models*

As with other evolutionary algorithms, GP can be described as a discrete-time Markov chain using transition matrices [Mitavskiy and Rowe, 2006, Poli et al., 2004]. This approach is much less developed than GP schema theory, and has only been applied to algorithms with more limited behaviour such as homologous crossover, based on the schemata defined by Poli and McPhee in developing a general schema theory. It is most likely the tractability of the problem that slows progress in developing such models.

4.9 EXTENSIONS TO GP

4.9.1 *Typing*

If we wish to have a compact representation, then we must introduce the notion of typing into our representation. Without typing, we cannot elegantly incorporate into our tree higher-level mathematical primitives such as matrices and computational elements such as loop constructs and storage access. There is a second use for typing mechanisms: by constraining the set of allowed structures, we may define our search space to exclude trees that we do not expect to be useful. For example, Montana [1995] gives one example where Strongly Typed GP reduces the order of the search space size from 10^{19} to 10^5.

If we try to restrict the structure of individuals without introducing a type system, then we are forced to "mend" or "punish" undesirable individuals created by our variation operators, or else constrain the operators themselves. This alternative may be simpler to implement, but it also inefficient and can strongly bias our search.

There are two ways of introducing typing to GP: the first is to use a grammar, which involves adopting a fundamentally different representation, for example in Grammatical Evolution [O'Neill and Ryan, 2003] or in work by Whigham and Rosca [1995]. The second is to use atomic and set-typing, introduced by Montana [1995] as Strongly Typed GP (STGP), which removes the requirement for closure of the function set.

STGP works by assigning each node and argument a type, and restricts population initialisation, crossover and mutation such that they obey type rules. Hence STGP ensures that only valid individuals are produced as a result of these operations. For example, in the initialisation process given in Algorithm 1 an extra constraint must be added to lines 3 and 5 that ensures the child is type-compatible. When selecting a child at line 5, we must also ensure that we take a path through the type system that will enable us to produce a tree of the requested depth. Strongly Typed GP is used when evolving programs in a subset of the C language in Chapter 7.

An interesting alternative method of allowing multiple types to be used without introducing problems in the evaluation process is to use an alternative form of evaluation such as in stack-based PushGP (see Section 4.10.1).

4.9.2 *Automatically Defined Functions*

ADFs were first suggested by Koza [1992]. One or more separate LISP trees (S-Expressions) termed "Automatically Defined Functions" are evolved as the lefthand subtree(s) of the individual, and the righthand subtree contains the value-returning branch. An example based on Koza [1992] is given in Figure 10. The ADF and main program tree together compose the even-4-parity function. The main tree is able to invoke the ADFs within its program and the ADFs have access to the same input variables. Thus ADFs allow the problem to be addressed using repeated references to the same subfunction, which may be parametrised. Intuitively, this method improves the scalability of GP as it removes the requirement that identical or similar subtrees be evolved at different points in the tree.

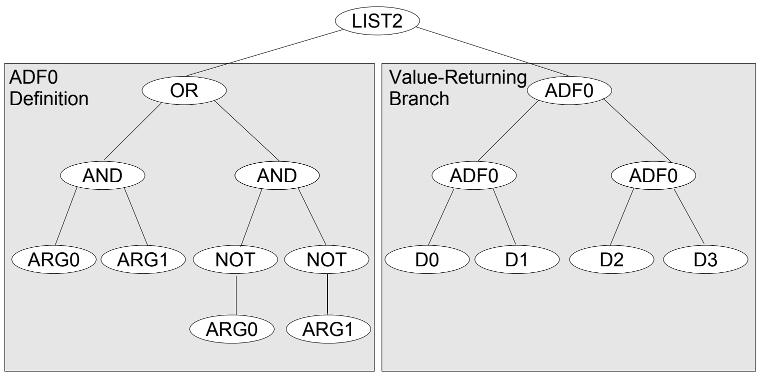


Figure 10: Even-4-parity function in a LISP S-Expression using an ADF.

ADFs are therefore particularly useful in solving problems where the solution is easily decomposed into subunits. For example, Koza demonstrated that solving Boolean problems such as the even-arity function can be achieved more time-efficiently by utilising ADFs.

An alternative approach to Koza's ADFs was described by Angeline and Pollack [1993], referred to as "module acquisition". A subtree is selected randomly from an individual during the run, and this is added to a library of such subtrees. The subtree itself is replaced by a reference to the library. The members of this library may then be referenced by other individuals in the population. Alternatively, under certain conditions a subtree is partially extracted such that it is parametrised. Automatically acquired modules may be "uncompressed", in that the library member may replace a reference to it and the individual expanded. This facilitates the return of genetic material to the population such that subcomponents of it may be shared with other individuals, i.e. it retains genetic diversity.

The various methods of implementing ADFs were subsequently generalised by Koza [1995a] into a taxonomy of what Koza described as *architecture-altering operations*.

4.10 ALTERNATIVE REPRESENTATIONS

Alternative representations have been utilised for application-specific purposes, and in some cases to alleviate some of the weaknesses of GP-based search. Traditional tree-based GP remains the most popular method, which reflects both the fact that alternative representations have as yet to provide any significant generalised benefit over tree-based GP, and also the simplicity of implementing a tree-based approach.

Examples include Stack-based PushGP [Spector and Robinson, 2002], CartesianGP [Miller and Thomson, 2000], Gene Expression Programming [Ferreira, 2001], Grammatical Evolution [O'Neill and Ryan, 2003] and Linear Genetic Programming [Brameier and Banzhaf, 2007]. Two examples are discussed below.

4.10.1 *Stack-based PushGP*

PushGP [Spector and Robinson, 2002] uses a linear representation and stack-based execution. This supports multiple data types (whilst preserving the validity of child programs) through the provision of type-specific stacks. Treating the code itself as a first-class data type, it allows code-manipulating operations such as the definition of variables and the creation of control structures and subroutines. The focus of PushGP is on what the authors term "autoconstructive evolution", or allowing self-adaptation of individuals through code-manipulating operators.

An individual is composed of a string containing data items and operators. Data items are pushed onto the appropriate stack, and operators pop their required arguments from the appropriate data stacks. The use of multiple stacks is in contrast to other stack-based GP systems such as FIFTH [Holladay et al., 2007], which ensures program validity by analysing the data types required to be on top of the stack before a code fragment executes (and those it leaves behind) in order to determine if two lists of instructions should be exchanged. Through the choice of a different representation, PushGP creates a rich set of features for a standard GP algorithm to exploit. However, this greatly enlarges the search space and it does not necessarily follow that providing the search algorithm with a potential method of achieving scalability means that the algorithm will be able to exploit that potential.

4.10.2 *Cartesian GP*

Cartesian GP [Miller and Thomson, 2000] uses a method that draws inspiration from electronic circuit layout, which is one of its key target domains. CGP originally used a two dimensional grid for layout of components that contain particular functions. A linear genome represents the interconnections of the nodes, and their functions. From left to right on the grid a program is divided into "stages". The interconnections between stages can be subject to constraints, such as restricting the distance (in stages) that the output of one function node can travel to the input of another. The result is that crossover and mutation operators are more intuitively appealing, in that they attempt to preserve context and support evolution in an incremental manner rather than the less subtle approach of GP subtree crossover.

Not all functional elements are necessarily used, and some units may be redundant and outputs left "dangling". There is an explicit amount of neutrality in this representation, an important concept in Biology and demonstrated to be advantageous on benchmark search problems [Miller and Thomson, 2000]. This method of interpreting an integer genome to create a phenotype that is guaranteed to be valid is similar to the grammar-based mapping in Grammatical Evolution [O'Neill and Ryan, 2003].

4.11 THE PROBLEM OF BLOAT

Bloat is an emergent property of GP, where an exponential or polynomial [Langdon, 2000a] increase in program size is observed in the absence of any improvement in fitness. Bloat creates large trees that are expensive to store, manipulate and evaluate. Therefore it is undesirable and a great deal of effort has been spent trying to understand and counteract the phenomenon.

This increase in program size corresponds to a similar increase in the amount of *introns* in the program tree. Introns are named after a term taken from Biology. In GP they refer to ineffective code, such as $(* \ x \ 1)$. This code has no effect on a program's overall behaviour but does contribute to its size and therefore the possible location of crossover points.

There are many theories of why bloat occurs, and no agreement as to which is correct, including the theories of hitchhiking, disruptive crossover, removal bias, search space bias and crossover bias. Langdon [1998] suggests that bloat is a general phenomenon particular to *"discrete variable length representations using simple static evaluation functions"*, caused by the crossover operator and the potential survival advantages that longer programs will have. The competing theories are discussed below.

4.11.1 *Disruptive Crossover*

The *disruptive crossover* (or "defence against crossover" or "replication accuracy") theory of bloat asserts that introns exist to provide protection for effective code from the disruptive nature of crossover. As discussed previously, crossover can be a disruptive mechanism and can destroy building blocks in a similar manner to GAs. However, building blocks in GP are likely to have a larger defining length than those in GAs and therefore are more prone to disruption. The effect of this disruption is that an individual will have a higher effective fitness if it is capable of avoiding the break-up of effective code. Through the inclusion of introns, there is a smaller probability that crossover will disrupt useful code. Therefore, a program's *effective fitness* i.e. its ability to survive and also replicate accurately, will be increased. Angeline [1994] was one of the first to observe that the disruptive nature of subtree crossover may be a prime cause of bloat in GP.

Reverting to a linear structure such as that used in Machine Code GP [Nordin, 1998] can constrain the relocation of genetic material. As individuals do not vary in shape, the crossover can ensure that genetic material is exchanged at the same points on the genome. However, the fragility of the representation remains, as a small change in one part of the program can have disastrous consequences for the semantics of the remainder of the program.

4.11.2 *Removal Bias*

Other theories view introns as a symptom of bloat, rather than the cause. The *Removal Bias* theory of bloat, suggested in Soule and Foster [1998], postulates that bloat is caused by the nature of subtree exchange. Removing a subtree from an area of the program that is inviable (it is not executed or has no impact on the program's behaviour) will have no impact on the program's fitness. Such subtrees are smaller than the program's overall size, and will usually be *much* smaller. Therefore, selecting a small subtree, rather than a larger one, is more likely not to reduce the fitness of an individual: there is a bias towards removing smaller rather than large subtrees. However, when inserting a replacement subtree into the position chosen, the size of the inserted subtree has no impact on the program's fitness (as it is in an inviable area), and over time selection will produce larger and larger trees.

4.11.3 *Search Space Bias*

The *Search Space Bias* (or "diffusion") theory [Langdon and Poli, 1997] states that there are more large-size highly-fit trees than there are small-size ones. Therefore the system will move away from smaller individuals. The *crossover-bias* theory of bloat similarly states that the disparity in solution quality between small and large trees will cause the larger produced by the crossover operator to prosper and thus increase average program size [Dignum and Poli, 2007].

4.11.4 *Bloat Control*

In order to counteract the emergence of bloat, practitioners can apply bloat control. The simplest is parsimony pressure [Koza, 1992], which introduces an extra component into the fitness function using a weighted sum as described in Section 5.4.1. This relies on a fixed trade-off between functionality and bloat prevention, which can be explored more fully using multi-objective optimisation [Jong and Pollack, 2003, Bleuler et al., 2001].

An alternative method is to change the algorithm itself, by biasing the crossover mechanism [Silva and Almeida, 2003], whereas perhaps the most precise approach is to control average program size based on size evolution equations, derived by Poli and McPhee [2008].

4.12 CONFIDENCE IN RESULTS

Heuristic search algorithms are a method of balancing solution quality and the time taken to find a solution. This allows us to approximately solve problems that cannot be solved by more conventional methods, but the trade-off is that we must be prepared to accept imperfect solutions. In some cases of automated programming, this is not acceptable, as the utility of a solution to a domain user may be Boolean. This is a major obstacle to the wider adoption of a method such as Genetic Programming in finding solutions. It is arguably an interesting psychological phenomenon that we are willing to accept human-written solutions that have been subject to less testing than to those created by GP. We may argue that humans are more methodical, that they arrive at the solutions through rational thought, but it is not clear that evolutionary systems are any less successful.

Confidence can be increased through testing, though testing cannot show the absence of bugs, and it is usually the case that we run the results of GP on a large validation set for this purpose.

Alternative directions are to produce artefacts that can be subject to model checking [Johnson, 2007], or formal proof methods [Chen et al., 2004]. For applications that have a Boolean level of acceptability, such approaches appear to be the most appropriate research directions.

4.13 COMPREHENDING GP OUTPUT

Related to the confidence we can have in the results produced by GP is the readability of the output solution. If it is not human-readable, then we cannot expect to manually verify its correctness or satisfactory design. There are three ways we can improve the readability of output:

- Constrain the search.
- Make readability an explicit goal of the search.
- Carry out post-processing in order to simplify solutions.

Haynes et al. [1995] argue that one application of Strongly Typed GP is to produce more comprehensible solutions, by restricting the structure of programs in the solution space. The role of bloat in increasing the obscurity of output is important, and parsimony measures have been shown to improve the readability of results [Jong and Pollack, 2003]. Post-processing involves removing inactive code, simplifying subexpressions and transformations. Smith and Bull [2007] apply both a parsimony pressure and post-processing to simplify the results of a set of data-mining problems using Genetic Programming.

4.14 NO FREE LUNCH

The *No Free Lunch* (NFL) theorem for search and optimisation is one of the most controversial theorems in the field of heuristic search. Wolpert and Macready [1997] introduced the NFL theorem in 1997 as an argument against the ability of heuristic search methods to generalise over all problems. What they proved was that any algorithm would perform equivalently over all possible problems, where a problem is defined by a mapping from the solution space X to the space of fitness values. However, due to the assumptions it makes, and the nature of problems of practical and commercial value, the NFL is of much less concern than was originally feared.

In particular, "the set of all problems" that the theorem reasons about must be a set closed under permutation and there are many counter-arguments to the existence of such situations in GP [Poli and Graff, 2009]. One example arises from the way that individuals are often evaluated over a series of test cases using the sum of squared errors as the fitness measure, which can prohibit the possibility of closure under permutation.

4.15 ACHIEVEMENTS OF GP

Initial interest in GP arose from its performance when compared to other artificial intelligence and machine learning methods on a range of standard benchmark problems such as the Santa Fe ant trail and symbolic regression. It also managed to solve these problems through application of the same method: the tree-based algorithm remained mostly the same, although determining the fitness function requires a problem-specific approach and can be a non-trivial task. This widespread applicability, as opposed to more specialised techniques such as neural networks, and the relative simplicity of the GP algorithm has encouraged its rapid adoption and experimentation across a wide variety of platforms, domains and applications.

Genetic Programming had by 2003 produced 36 "Human Competitive" results detailed by Koza et al. [2003], and many more since at the GECCO Hummie awards. Koza defines human competitive by a number of criteria that places GP in direct competition with the previous work of designers and engineers. This demonstrates a focus on novelty and invention through the use of search, rather than the specific types of results achieved. The majority of the problems are circuit layout problems rather than programs, and this supports Fogel and Atmar [1990]'s observation that the applicability of GP (and evolutionary, crossover-based programming in general) may be limited to a subset of problems that have a low degree of epistasis.

Koza et al. [2003] discusses the relationship between Moore's Law and the number of human-competitive results in GP. The relationship is such that the results of applying GP is improving directly with the increase in computing power. However, the implication of this trend is that the technique of GP itself is not improving.

4.16 COMPARING GP TO RANDOM SEARCH

With a great deal of foresight, Fogel and Atmar [1990] criticised the method of evolutionary search using crossover in machine learning applications where behaviour-defining structures undergo genetic manipulation. Their comparisons to mutation-based search demonstrated there was no advantage to using crossover. Their conclusion was that this may limit the applicability of crossover-based search to specific applications that have a low degree of inter-dependence between subcomponents of a solution.

More specifically, Luke and Spector [1997, 1998] compared GP to mutation-based search. Their mutation operator replaces a randomly selected subtree with a randomly generated one. They found standard GP offered little improvement over mutation-only search on a range of problems from the GP literature. Furthermore, they found a large dependency on the choice of parameters in order to ensure this small improvement was achieved. Similarly, Angeline compared GP to "Headless Chicken Crossover" [Angeline, 1997], whereby a randomly constructed second parent is generated to participate in crossover. He found similar results to those of Luke and Spector. I have recently applied rigorous experimentation to this comparison and found that crossover is beneficial for some of the problems investigated [White and Poulding, 2009].

4.17 OPEN ISSUES

4.17.1 *Theory of Bloat*

No agreed explanation for the phenomenon of bloat in a GP run has been proposed. This problem appears amenable to experimentation, and with a systematic methodology it may be possible to eliminate some of the theories of bloat. It is clear that the use of crossover directly or indirectly causes bloat, and by varying the rules of this operator it may be possible to eliminate some of the proposed explanations.

4.17.2 *Scalability*

Much of the GP literature has focused on small examples as a basis for theoretical study, experimentation of extensions to GP, or comparison to proposed alternative algorithms. In particular, amongst researchers there is a concern about the reach of GP: beginning at a given level of abstraction, how much can GP achieve for itself? For example, given machine code instructions as its function set, will it be able to solve the same problems that it would if Fast Fourier Transforms and vector operations were included in the function set? This is actually two questions: will the algorithm *ever* be able to solve the problem without the higher-level functions made available to it? Will it be able to solve the problem in an acceptable amount of time?

4.17.3 *Theoretical Basis and Justification*

As we have seen, there have been several attempts to devise a general schema theorem for GP. Whilst progress has been made, it is not clear that these results provide a formal explanation as to why GP should be selected as a method for program space search. The outstanding problem is to demonstrate both empirically and theoretically that building blocks exist in GP regardless of the problem domain, that they are the rule rather than the exception, and that subtree crossover is the most effective way to exchange these building blocks.

4.18 SUMMARY

In this Chapter, I have given an overview of the field of Genetic Programming. In the previous chapter, I outlined methods that can be used to optimise the non-functional properties of embedded systems. What remains now is to combine the two: to examine applying Genetic Programming to control the non-functional properties of software. There is a related and well-established line of research in this area: the control of bloat (see Section 4.11), although we do not usually treat bloat itself as a program characteristic: it is an artefact of the search algorithm. Bloat presents a second objective for the search, casting our problem as a multi-objective optimisation (MOO) problem. All optimisation considering multiple properties of a solution can be formulated as a MOO problem, and therefore at this point it is necessary to introduce the field of MOO, which I do in the next chapter.

5 MULTI-OBJECTIVE OPTIMISATION

5.1 INTRODUCTION

Low-resource systems design involves multiple conflicting constraints, and multi-objective optimisation (MOO) methods offer the ability to find solutions that satisfy and trade-off such multiple criteria. In this chapter, I give a general review of MOO methods and also look at previous applications of MOO techniques relevant to the work presented in this thesis.

These techniques fall broadly into two categories: aggregation and Pareto-based methods. Pareto-based methods are usually implemented using evolutionary algorithms, which is part of the motivation for adopting these methods in this thesis. A detailed discussion of the strengths and weaknesses of these methods is given by Freitas [2004], who finds that the Pareto-based approaches are more principled than aggregation methods, which tend towards trial-and-error.

The most cited texts on multi-objective evolution are Coello Coello et al. [2002] and Deb [2001].

5.2 PROBLEM DEFINITION

Multi-objective problems are defined by a set of objectives represented by multiple fitness functions $f_1, f_2, \ldots, f_k$, and a series of constraints that restrict the search space to $C \subseteq X$, a feasible subset of X. For example, Figure 11 illustrates a feasible region within the solution space, where a solution p is a pair of parameter values x_1 and x_2. The feasible region of a problem is not necessarily continuous.

Coello Coello [2000] defines multi-objective optimisation as optimising a vector-valued fitness function with n components:

$$\mathbf{F}(p) = (f_1(p), f_2(p), \ldots, f_n(p)) \tag{5.1}$$

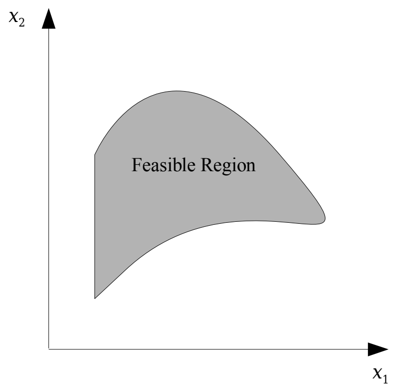


Figure 11: A feasible region within a solution space.

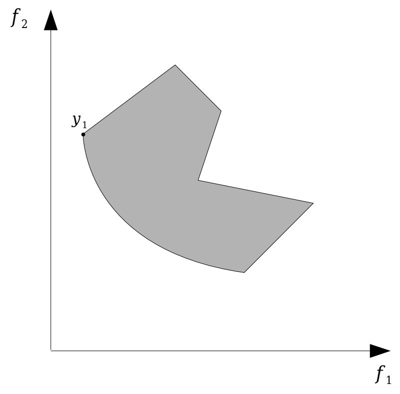


Figure 12: Visualising trade-offs in objective space.

subject to m inequality constraints:

$$g_i(p) \geq 0,\ i = 1, 2, \ldots, m \tag{5.2}$$

and q equality constraints:

$$h_i(p) = 0,\ i = 1, 2, \ldots, q \tag{5.3}$$

A multi-objective search algorithm must attempt to optimise each component of the fitness vector, and explore the trade-offs between components. In a low-resource system, the objectives are both functional and non-functional requirements. Where they conflict, we see trade-off spaces as in Figure 12.

5.2.1 *Objective Space*

The range of **F** can be visualised in *objective space*. Each feasible point $p \in C$ in solution space is mapped to a point y in the objective space. Trade-offs between objectives can be visualised by their location in this space. For example, in Figure 12, y_1 minimises objective f_1 at the expense of f_2, and the two objectives are conflicting. Automated exploration of this space may aid an engineer in assessing where the design limits are, and the nature of the trade-offs involved. The gradient of the relationship may indicate where most is to be gained from sacrificing one objective for another.

5.2.2 *Separation of Constraints and Objectives*

In low resource system design, it is usually the case that non-functional requirements can be decomposed into a constraint and an objective [Dick and Jha, 1997]. For example, if the system CPU has already been specified, then it is essential that the software solution be efficient enough so that scheduling the given set of tasks on the processor is feasible. It may also be *desirable* that the software use fewer CPU cycles and hence reduce its power consumption.

Constraints can be satisfied using a penalty fitness function component, and some multi-objective algorithms treat them entirely separately [Aguirre et al., 2004].

5.3 types of multi-objective optimisation

There are two major classes of multi-objective search methods. Firstly, those often described as *classical* methods, which I refer to here as *aggregation-based* methods. Secondly, Pareto-based methods that utilise the concept of Pareto non-dominance to find a set of competing solutions. Aggregation-based methods are limited in that they do not consider more than a single solution at a time. Any method or tool that provides an engineer with a single solution in the face of conflicting trade-offs restricts the information available. A compiler is typically such a tool: it has an implicit bias in the solutions it will generate.

5.4 aggregation of objectives

Aggregation techniques attempt to aggregate the separate requirements of a problem into a single overarching objective. The fitness function **F** is collapsed into a scalar-valued function. As such, aggregation methods attempt to find a single solution that is in some sense optimal. Here I describe three of the most popular aggregation methods.

5.4.1 *Weighted Sum*

The most common aggregation technique is the weighted sum approach, using a fitness function specified as follows:

$$F(p) = \sum_{k=1}^{n} w_k \cdot f_k(p) \tag{5.4}$$

where

$$\sum_{k=1}^{n} w_k = 1 \tag{5.5}$$

Often, the objective values themselves are normalised to lie within $[0, 1]$. The approach of a simple weighted sum fitness function is easily implemented and is commonly used where there is a clear priority between objectives, or the objectives conflict to a small or no extent. The relative weightings of the different objectives must be decided, i.e. the weightings must be normalised. This is often difficult or impossible to achieve in the embedded domain and can involve changing political and economic factors [Berntsson Svensson et al., 2009]. For example, how many units of power saved can be described as equivalent in cost to a quantified increase in response time? This may lead to experimentation in weighting, which is computationally expensive.

Weighted sums define a single maximum value for an individual's fitness. Automated optimisation methods will attempt to locate this value and, if successful, the population will converge to this optimum. In the bounds of the search space however, one program may minimise processor time required but use a large amount of memory. Another solution may minimise memory usage but create more communication overhead. The opportunity for an engineer to make comparisons between such alternatives is lost.

The weighted sum approach cannot locate all of the possible levels of trade-off between objectives for some problems, and it cannot converge

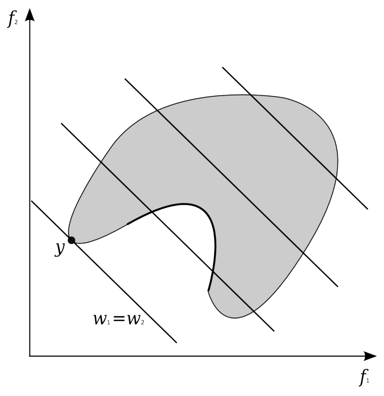


Figure 13: Limitations of the weighted sum method.

on a solution it if lies on the surface of a concave region in objective space, such as that highlighted in Figure 13. In this space, for a given fitness value we can plot a line with gradient $-w_1/w_2$, a contour linking all points with that value. We wish to minimise this value, so we can see that the optimal point for any given set of weightings will be the point that lies on a tangent to the feasible region.

Figure 13 shows the optimal point y that will have the best fitness value when $w_1 = w_2$ i.e. when the weightings are equally balanced. By varying the relative weightings of the two objectives, different solutions will be considered optimal. As no tangent can exist that passes through the points in the concave region, these points cannot be found regardless of the weightings used. However these points are potentially desirable, as they represent unique trade-offs between the two objectives. A rigorous argument of this limitation is given by Das and Dennis [1996].

5.4.2 *Goal Programming*

Goal programming is a long-established set of methods that direct the search to a designated aim, such as a fixed target value T_i for each objective. The algorithm then attempts to satisfy these goals, even if this results in a suboptimal solution. This method was introduced over half a century ago by Charnes et al. [1955].

The goal for each component of the fitness vector can take the form of an exact value or an inequality, where an inequality will specify an acceptable range of values for that fitness measure. The fitness of an individual is measured by the distance between its current fitness vector and the goal fitness vector. The objective is to optimise this distance, a simple example being to minimise:

$$\sum_{i=1}^{k} |f_i(p) - T_i| \tag{5.6}$$

Different distance metrics are used in different implementations. If the goal is to attain a fitness less than (or more than) a particular value, the *deviation* is the distance from achieving this goal, or zero if the goal is achieved. An interval can be treated as a pair of goal inequalities.

These multiple deviation values must then be aggregated into an overall fitness function, which faces the same problems as any other aggregation method, such as deciding upon suitable weight values or priorities between objectives. The method may lead to a suboptimal solution as the algorithm will only attempt to satisfy the goals supplied.

Goal programming assumes a decision maker can set targets for optimisation. In low-resource systems we often wish simply to know the extent to which we may reduce resource requirements. Goal programming does not provide an estimation of what is possible, but will only try to achieve the goals set by a designer for a particular problem.

5.4.3 *ϵ-Constraint Method*

The ϵ-constraint method first proposed by Haimes et al. [1971] overcomes some of the limitations of a weight-based summation by considering each individual fitness component in turn. Whilst component f_i is being optimised, all $f_k, k \neq i$ are subject to a set of supplied constraints. Thus the aim of the algorithm is then to find the minimal value for f_i whilst working within the restrictions of the constraints on the other fitness components. A vector of ϵ-values is used to effectively constrain the search to a region of the objective space. Each search within a particular fitness component attempts to finds a Pareto-optimal solution (see Section 5.5.1), such that repeated searches can be used to enumerate the Pareto-front.

The process of individually exploring each fitness component is computationally expensive, although less so than an exploration of the space of all possible weights when using a weighted-sum method. The engineer must provide a vector of constraint values, and this selection will affect the quality of solutions that the method can discover. It places extra burden on the decision-maker to provide information that may not be available.

5.5 PARETO-BASED OPTIMISATION

5.5.1 *Pareto Fronts*

Pareto-based optimisation searches for multiple solutions rather than a single point. Unlike aggregation methods it does not require that the fitness vector of an individual be reduced to a single scalar value. The search algorithm aims to discover a surface in the objective space known as a *Pareto front*. When objectives conflict, these methods output a set of solutions offering the opportunity for comparison.

The Pareto front is defined by the concept of *Pareto non-dominance*, which relates one solution to another in terms of each fitness component, and is denoted $p_1 \preceq p2$. Solution $p_1 \in C$ dominates solution $p_2 \in C$, that is $p_1 \preceq p_2$ iff:

$$\forall_i : f_i(p_1) \leq f_i(p_2) \wedge \exists_j : f_j(p_1) < f_j(p_2) \tag{5.7}$$

The Pareto-optimal set is composed of individuals that are non-dominated. A solution p_n is non-dominated iff:

$$\neg \exists k : p_k \preceq p_n \tag{5.8}$$

Intuitively, solution p_n is non-dominated if there does not exist an alternative solution p_j that has both a better (lower) value for one or

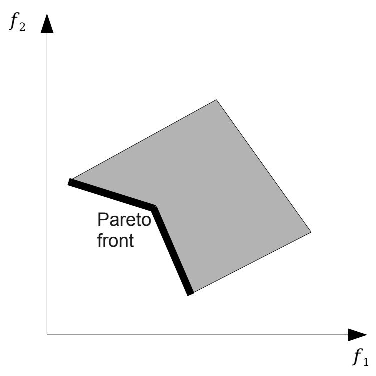


Figure 14: An example Pareto front in objective space.

more of the components of the fitness vector, and no worse (higher) value for any of the other components.

Figure 14 illustrates a Pareto front in objective space. The front contains all non-dominated solutions, i.e. all points where an improvement can be made to one fitness component only at the expense of another fitness component. This illustrates the main limitation of aggregation-based methods: they can locate at most only a single point on the front.

It is not usually possible to predict in advance where the Pareto front will lie, and it is not possible to know if a given set of solutions accurately describes a Pareto front for the problem. As a result, Pareto-based algorithms attempt to continually improve an approximation to this front.

5.5.2 *Elitism*

As described by Zitzler et al. [2001], elitism is a crucial component of any evolutionary multi-objective optimisation algorithm. Elitism is the process whereby solutions can survive from one generation to another without undergoing manipulation through genetic operators such that its genome remains intact. Multi-objective evolutionary algorithms usually implement this technique by retaining a separate set of non-dominated individuals distinct from the main population. Elitism is effective because it prevents the loss (break-up) of a non-dominated individual through genetic manipulation.

Some problems have Pareto fronts with sparsely populated areas, i.e. where there exist few solutions that achieve certain trade-off values. In order to retain these solutions, it is necessary to use elitism to ensure these "lonely" individuals survive, as it is the group of non-dominated solutions that *together* compose the estimation of the Pareto front.

5.5.3 *Selection and Niching*

As an evolutionary algorithm searches the solution space, it will find progressively fitter individuals. In the case of Pareto-based optimisation, the algorithm must discover a surface in this space. The danger of selection-based methods is that they may strongly favour an individual that represents a step improvement over the current population. This will then encourage the collapse of the front as the inferior, yet potentially useful, solutions that are located in other parts of the search space are removed by their lower performance. Even without this fitness pressure, populations may converge to a small area of the objective space due to genetic drift. It must be ensured that the search does not converge to a single area of the Pareto front, but instead maintains a set of individuals that incrementally progresses across the objective space.

If we are to explore the variety of trade-offs available when creating low-resource software, we must maintain a set of solutions. The composition of the set will harbour useful information that a single solution cannot. For example, the location of "sweet spots", that is points of inflection in the relationship.

Maintaining this diversity of points within the current estimate of the Pareto front is achieved in two ways. Firstly, by assigning fitness values in terms of relative performance of an individual over the population through Pareto dominance. Secondly, by punishing solutions that are closely co-located in the trade-off space through a density function that reduces the fitness of individuals in heavily populated regions.

Individuals are ranked (the exact manner depends on the particular algorithm involved) according to their non-dominance within the population. Usually, the population is divided into "fronts", where non-dominated individuals are repeatedly selected within the population, allocated a ranking and then removed from consideration. This ensures the survival of weaker individuals, provided they can perform well on some fitness components or find less explored areas of the search space. The application of this concept is referred to as *niching*, and is also used elsewhere in evolutionary computation to favour exploration (usually of the search space) rather than exploitation.

Niching works by reducing the probability that an individual in a densely populated region of the objective space will be selected to be a parent. Sareni and Krahenbuhl [1998] generalise this idea as an adjustment of fitness thus:

$$f'(p) = \frac{f(p)}{m(p)} \tag{5.9}$$

where m is a niching factor calculated using a sharing function that estimates the crowding in a given area of the search or objective space:

$$m(p) = \sum_{p' \in P} s(p, p') \tag{5.10}$$

The sharing function s is based upon some distance metric such as tree edit distance in the Genetic Programming search space, or Euclidean distance in objective space. The two MOO algorithms detailed in Section 5.6 employ forms of sharing.

Alternative methods of niching exist, though they are in general much less popular than fitness sharing. For example restricting the crossover of individuals to those within a certain proximity, encouraging speciation amongst the population. The expense of such niching is often in computational complexity, as individuals must be widely compared.

5.6 PARETO-BASED ALGORITHMS

There have been many different implementations of Pareto-based techniques, and here I describe two algorithms representative of those

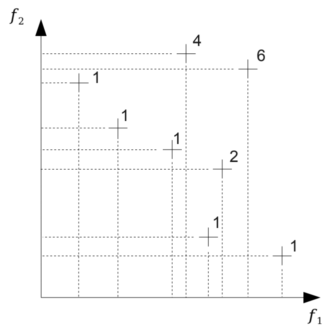


Figure 15: MOGA rankings within the objective space.

proposed. Many individual algorithms are in fact modifications or optimisations of previous techniques. In this thesis I use SPEA2, because it has previously been successfully applied to related MOO problems in combination with Genetic Programming [Weise and Geihs, 2006].

No algorithm is guaranteed to converge to an exact representation of the Pareto front, only an approximation of it. The efficacy of a particular Pareto-based algorithm in locating this front is dependent on the nature of the objective space and the parameters of the algorithm.

5.6.1 *MOGA*

The Multi-objective Genetic Algorithm (MOGA) [Fonseca and Fleming, 1993] is one of the original multi-objective evolutionary algorithms in the literature. Fonseca and Fleming focus on the role of multi-objective algorithms in providing a set of alternative solutions to a decision-maker, and suggest modifications to facilitate the input of guidance and feedback from the decision-maker into the algorithm. MOGA gives each individual a ranking as follows:

$$rank(p_i, t) = 1 + d_i^t \tag{5.11}$$

The ranking of an individual p_i at generation t is 1 if it is non-dominated, otherwise it is 1 plus the number of individuals it is dominated by in the current population. Figure 15, based on Fonseca and Fleming, illustrates rank assignments within the objective space. Not all values of rank will necessarily be attributed to one or more individuals in any given population.

Algorithm 5 illustrates how MOGA calculates fitness. Each component within the fitness vector of an individual is evaluated. The ranking of each individual is calculated, and then interpolated. The fitness values of individuals with the same rank must be averaged, to ensure they will be sampled at the same rate using fitness-based selection. An example interpolation step given by Deb [2001] is as follows:

$$f(p_i) = N - \sum_{k=1}^{r_i - 1} \mu(k) - 0.5(\mu(r_i) - 1) \tag{5.12}$$

Here, N is the number of objectives, r_i is shorthand for $rank(p_i)$, the rank of the individual, and $\mu(k)$ is the number of solutions of rank k.

Algorithm 5 Pseudocode for MOGA Fitness Assignment.

```
1: $\forall p_i \in P_t$ evaluate $\mathbf{F}(p_i)$
2: $\forall p_i \in P_t,\ d_i^t \Leftarrow |\{p_j : p_j \in P_t \wedge p_j \preceq p_i\}|$
3: $\forall p_i \in P_t,\ \text{rank}(p_i, t) \Leftarrow 1 + d_i^t$
4: $\forall p_i \in P_t, f(p_i) \Leftarrow$ interpolate $rank(p_i, t)$
```

To maintain the diversity of solutions and accurately estimate the Pareto front, a niching mechanism is used. The MOGA method uses *fitness sharing*, which creates selection pressure to maintain diversity through a component in the fitness function. The level of niching required must be set using a niching size σ_{share}. Niching is performed over the objective space, and σ_{share} defines the limiting distance two individuals can be apart before their fitness is affected by the niching mechanism. This presents a problem if the practitioner does not have an accurate expectation of the nature of the Pareto front. For example, Fonseca and Fleming estimate the value of σ_{share} required based on the size of the trade-off surface. However, using their method would require knowledge of the extreme values of the front – and wishing to discover such extremes is often part of the original motivation of applying MOO methods.

Rodriguez-Vazquez et al. [1997] present a multi-objective Genetic Programming algorithm (MOGP), which is based on MOGA. As MOGA uses fitness space-based ranking and niching, these concepts can be applied directly to Genetic Programming. They apply MOGP to system identification problems with up to seven distinct objectives.

5.6.2 *SPEA2*

SPEA2 [Zitzler et al., 2001] is the revision of the Strength Pareto Evolutionary Algorithm. The algorithm maintains both a general population and a fixed-size archive, which contains the non-dominated individuals found so far. The algorithm employs fine-grained fitness assignment to carefully preserve diversity within its current solution set.

An individual's fitness is determined by both Pareto-based ranking and density information. The strength $S(p_i)$ of an individual is defined as the number of others that it dominates:

$$S(p_i) = |\{p_j \ : \ p_j \in P_t \cup \bar{P}_t \wedge p_i \preceq p_j\}| \tag{5.13}$$

where $\bar{P}_t$ is the archive, and $\preceq$ is the Pareto-dominance relation. The raw fitness of an individual is the strength of the individuals it is dominated by:

$$R(p_i) = \sum_{p_j \in P_t \cup \bar{P}_t, p_j \preceq p_i} S(p_j) \tag{5.14}$$

The second component of the fitness function is the density estimation of the individual, given by the distance from a solution p_i to its *kth* nearest neighbour, σ_i^k. The density component is:

$$D(p_i) = \frac{1}{\sigma_i^k + 2} \tag{5.15}$$

The fitness of an individual is then:

$$F(p_i) = R(p_i) + D(p_i) \tag{5.16}$$

The archive for the next generation is generated based on this combined fitness measure. Only those non-dominated individuals that are sufficiently dispersed across the current front are allowed to enter the selection process, i.e. only the individuals in the archive are allowed to mate. Algorithm 6 gives the pseudo-code for the algorithm.

The run-time of SPEA2 is dominated by the niching calculation, and has complexity $O(M^2 \log M)$ where $M = |P| + |\bar{P}_t|$, the combined size of the population and archive.

Algorithm 6 Pseudocode for SPEA2.

```
1: $t \Leftarrow 1$
2: archive $\bar{P}_t \Leftarrow \emptyset$
3: initialise general population $P_t$
4: loop
5:     for all $p_i \in P_t \cup \bar{P}_t$ do
6:         evaluate objective values of each individual $p_i$
7:     end for
8:     for all $p_i \in P_t \cup \bar{P}_t$ do
9:         $S(p_i) \Leftarrow |\{p_j \ : \ p_j \in P_t \cup \bar{P}_t \wedge p_i \preceq p_j\}|$
10:    end for
11:    for all $p_i \in P_t \cup \bar{P}_t$ do
12:        $R(p_i) \Leftarrow \sum_{p_j \in P_t \cup \bar{P}_t, p_j \preceq p_i} S(p_j)$
13:    end for
14:    for all $p_i \in P_t \cup \bar{P}_t$ do
15:        $D(p_i) \Leftarrow 1/(\sigma_i^k + 2)$
16:        $F(p_i) \Leftarrow R(p_i) + D(p_i)$
17:    end for
18:    $\bar{P}_{t+1} \Leftarrow \{p_j : p_j \in P_t \cup \bar{P}_t \wedge \forall p_k \in P_t \cup \bar{P}_t, \ p_j \preceq p_k \}$
19:    if $|\bar{P}_{t+1}| <$ archive size then
20:        fill archive with dominated individuals
21:    end if
22:    if $|\bar{P}_{t+1}| >$ archive size then
23:        truncate archive based on density
24:    end if
25:    $t \Leftarrow t + 1$
26:    if termination criteria satisfied then
27:        return $\bar{P}_t$
28:    end if
29:    create $P_t$ from $\bar{P}_{t+1}$ using selected algorithm
30: end loop
```

5.7 COEVOLUTION

Coevolution is the process of evolving multiple populations simultaneously, where each population solves a different but inter-related problem. Coevolution has three main applications:

- It can be used in combination with Pareto-based optimisation to satisfy multiple objectives in a scalable manner.

- It can be employed to improve the performance of algorithms in finding generalising solutions and identifying difficult test cases.
- If a solution can be decomposed into several distinct parts then coevolution can be used to produce compatible components. For example, a compression and a corresponding decompression algorithm.

Coevolution is either competitive or cooperative. Under competitive evolution, the fitness of one population is defined by its relative performance compared to another, much like predator-prey relationships in the physical world. In cooperative evolution, separate populations work together to solve a problem, for example by dividing a task into subproblems and solving those problems individually. Both methods have applications within multi-objective optimisation.

Coevolution is used in this thesis to ensure that solutions maintain functional generality when optimising their non-functional properties. This is important because including non-functional properties in fitness measurement offers an individual solution the opportunity to survive by discarding functionality. Test cases must be capable of exposing the omission of functionality, and coevolution increases the fitness pressure on the evolving solutions.

5.7.1 *Competitive Evolution of Test Cases*

Competitive evolution can be used to coevolve a population of test cases alongside the solution population. The solution population is tested against the test population (or a sample).

This approach can be used in situations where exhaustive testing is not possible, and it is not clear what subset of test cases will achieve the best results. Alternatively, if it is required that only a subset of test cases are used (in order to reduce the computational cost of testing and execution time) competitive evolution can find a suitable subset. Competitive evolution may also be used in order to discover particularly difficult test cases for a given set of potential solutions. This avoids the problems of over-fitting test data and unacceptably poor performance on pathological test cases. As Angeline [2000] points out, the real advantage to using a competitive evolutionary approach is that it scales well: the test cases will evolve to stretch the current solutions as much as necessary.

Coevolution of test cases was used by Hillis to evolve minimal sorting networks [Hillis, 1990]. Whilst he successfully produced correct sorting networks without coevolution, he was unable to match the best results designed by human engineers using a simple objective of minimising the number of connections. The most challenging test cases for a candidate solution were dependent on the structure of the candidate itself. Test cases were therefore coevolved with the solution population and found to improve the results of the algorithm. This motivates my use of competitive coevolution in Chapter 7.

One weakness of this approach is the task of ensuring that the two populations evolve in synchrony [Miconi, 2009]. If the test data population becomes too difficult before the algorithm has found solutions that can at least solve some of the test cases, then the fitness of individuals in the population will be indistinguishable, and no information will be available to guide the search. Similarly, genetic drift or other bias in the search method may lead to the state of *disengagement*, where the two populations are no longer reacting to each other but instead following a direction determined by other selection pressures. At the other extreme, the two populations may become closely intertwined so that the solutions evolve only to solve the coevolved test cases and no longer generalise over the remainder of the test case space.

5.8 multi-objective genetic programming

Combining MOO and GP is a central component of this thesis, and there has been a certain amount of work in this area in the past. By far the most common of such applications is the use of MOO to encourage parsimony and thereby avoid the emergence of bloat in GP [Bleuler et al., 2001]. This has also been used to encourage generalisation and to improve the comprehensibility of output. Jong and Pollack [2003] carried out a study of bloat control methods on example problems, but also note the importance of diversity-preserving techniques to ensure bloat control is not exerted at the cost of exploration of the search space.

Perhaps the only work that uses MOO to improve *existing programs* to achieve desirable non-functional properties is that on program compression by Langdon and Nordin [2000], where the authors attempted to reduce the size of existing programs using GP. They use a multi-objective approach to control program size, having started with existing solutions in a similar manner to the seeding used in Chapter 7. They applied this approach to classification and image compression problems. They were particularly interested in the impact such a method would have on the ability of final solutions to generalise.

Weise and Geihs [2006] used GP to evolve an election algorithm for a wireless sensor network, a similar platform to those targeted here. They use a fitness function with a parsimony component as well as explicitly specifying minimal use of memory and communication as objectives. This very relevant work achieved initial results but no follow up has been published.

Applications with many objectives can be found in Koza's work in evolving analogue circuits from the late 1990s onwards. For example, evolving an amplifier circuit [Koza et al., 2004a] with 16 different desired properties, such as gain, supply current and offset voltage. This required multiple types of evaluation for each individual, and overall fitness was calculated by a weighted sum of the components.

5.9 summary

In this chapter, I have introduced the fundamental concepts of multi-objective optimisation, the use of coevolution in a similar context, and examples of relevant work combining MOO with Genetic Programming. After Chapter 3 introduced the problems of engineering low-resource embedded systems, this and the previous chapter have outlined methods for attacking those problems. Over the next three chapters, I apply these techniques to example problems in the embedded domain, demonstrating the potential of these search and optimisation algorithms to explore design spaces in powerful new ways.

Part III

EXPERIMENTATION

6 EVOLVING RESOURCE-EFFICIENT SOFTWARE

6.1 INTRODUCTION

A common form of non-functional requirement in embedded systems is the demand to minimise a property such as power consumption. Power consumption has become of increasing importance as modern deployment paradigms such as Wireless Sensor Networks [Akyildiz et al., 2002] and SmartDust [Warneke et al., 2001] demand the miniaturisation of the system, including the battery. Consequently, battery capacities are small and harvesting energy from a system's physical environment is the object of intensive research [Paradiso and Starner, 2005]. In this chapter, I investigate the ability of Genetic Programming to find solutions with low power requirements, creating software that aids the system goal of reducing resource consumption.

When selecting an example application for this experimentation, there were two goals in mind. Firstly, the application must be software that is typically executed with high frequency within an embedded system, such that its power consumption contributes significantly to overall system power usage. Secondly, it must be small enough to allow large experimental evolutionary runs to take place on a single PC. The choice made in this chapter is to evolve low power pseudorandom number generators (PRNGs). PRNGs are a common component in embedded systems, for example to be used in communication protocols or security applications, and as such they are of practical interest to a developer.

In reducing power consumption, there is often an explicit trade-off between performance and consumption, or even functionality and consumption. Rather than simply considering "this is what we must do: minimise the power required", we can consider a potentially more useful question: "given this much power, how well can we achieve our functional goals?". Thus in this chapter we consider the trade-offs between the quality of PRNGs against the power that they consume.

A combination of MOO and GP is used to explore solution and objective spaces, both to generate PRNGs and to provide insight into the trade-offs that can be made between different objectives. Specific uses for such an approach are:

- Satisfying exact constraints specified as a requirement.
- Finding the extent to which objectives may be balanced: is the number of distinct individuals in the Pareto front large or small?
- Identifying how many different distinct values of each objective are represented within the Pareto Front, i.e. what level of granularity of trade-off exists?
- Empirically quantifying the relationships between multiple objectives: how much energy can be saved by increasing the available processing time, for example?

- Finding a set of programs offering multiple different trade-offs. A solution may be chosen from this set depending on future requirements, whether statically at design-time or dynamically at run-time. For example, a new target platform may have fewer resources, or a different mode of operation may reduce the amount of available processing time.

This chapter is based upon work in White et al. [2008].

6.2 PSEUDORANDOM NUMBER GENERATORS

PRNGs are important program components that generate a stream of pseudorandom numbers, where the precise definition of "random" is dependent on the way in which the output will be used [Knuth, 1997]. PRNGs are often found in embedded systems. For example, they may be used for key generation in cryptographic applications or within communication protocols for collision resolution. Typically, PRNGs are small code fragments that are frequently used and thus random number generators with specific non-functional properties are a useful tool.

PRNGs have been produced using both GP [Koza, 1991, Jannink, 1994, Hernandez et al., 2004, Lamenca-Martinez et al., 2006] and other bio-inspired techniques (for example, [Sipper and Tomassini, 1996]). Where heuristic search techniques have been applied in the past, the key difference between alternative methods has been the fitness function selected to establish how good a candidate PRNG is. Knuth [1997] gives an extensive review of tests to measure the "randomness" of a sequence, that are potential candidates for a fitness measure, although more stringent tests are required for cryptographic primitives. No previous work has explicitly targeted low power as an objective when searching for PRNGs.

6.2.1 *Producing PRNGs with GP*

Koza [1991] demonstrated the ability of GP to evolve pseudorandom number generators, outputting a stream of binary digits when given as input a sequence $1, 2, 3, \ldots, 2^{14}$. He used an entropy-based measure that summarised the measurement of the distribution of possible subsequences. By this measure, for a sequence of N integers where each integer can take k different values, the desired probability of each possible subsequence occurring is $\frac{1}{k^N}$. Koza's experiments successfully produced individual programs that provide sequences with high entropy, and perform well when measured against commercial randomisers under two statistical methods from Knuth [1997]. Koza notes that in some sense the distributions are too perfect, in that the divergence from an ideal distribution is so small that the output could be considered unlikely to be from a truly "random" source.

Jannink [1994] used GP to predict the outputs of existing commercial generators, as a measure of their quality. He then took a similar approach to creating new randomisers, by using coevolution (see Section 5.7) to competitively evolve generators and predictors, thus using competition to determine fitness in place of statistical measures of randomness. The function set provided to GP was similar to that used in this chapter, though Jannink included memory reading and writing operations. Evaluating the success of the evolved generators against standard battery tests was not included in the paper, and therefore it is difficult to make comparisons between the effectiveness of this coevolutionary approach and those that define fitness using statistical measures.

Hernandez et al. [2004] also applied GP to PRNG creation. Part of the stated aim of their work was to consider not only the functionality of PRNGs but also the efficiency of the evolved solution, and hence this work is closely related to the aims of this chapter in satisfying non-functional requirements. This work was continued and expanded upon by Lamenca-Martinez et al. [2006]. Much of the experimental work reported here is based on this later work. In particular, the fitness measure and choice of function set are taken from these papers. The fitness function used measures the nonlinearity of a PRNG's output, as an alternative to statistical measures of randomness. Further details of the fitness function are given in Section 6.4.

In order to produce efficient PRNGs, Hernandez et al. [2004] and Lamenca-Martinez et al. [2006] restricted the function set to contain operations that could be executed quickly, and attempted to evolve a "minimalist's PRNG". Direct measurements of efficiency were not made, and both papers comment on whether the inclusion or exclusion of a MULT (multiply) function would be appropriate due to its relative expense. This chapter extends their work by explicitly examining a non-functional property of the solutions: power consumption. It also provides a method of comparing the impact of the MULT function on efficiency-functionality trade-offs for a specific target architecture.

6.3 power simulation

In this work, I make use of the SimpleScalar simulator [Burger et al., 1996] in combination with the Wattch power consumption model [Brooks et al., 2000]. I use an unmodified version of the simulator, which targets the 64-bit PISA instruction set, a similar design to the MIPS-IV ISA [Price, 1995]. The simulator's default settings include two 8kB level 1 cache banks, one for instruction and one for data, and a unified 256kB level 2 cache. The specific choice of architecture does not concern us too much: only that it is consistent in relative measurements between two individuals, as we are interested in designing a generalisable method independent of target platform.

System calls are intercepted by the simulator, so that without loading an operating system it is possible to capture functional behaviour.

Wattch is a cycle-level power simulator based on a parameterised processor model, and provides overall power estimates for a program's execution, calculating power consumption P_d for different logic units using the following equation:

$$P_d = CV_{dd}^2 af \tag{6.1}$$

C is the load capacitance, V_{dd} the supply voltage, f the clock frequency and a an activity measure that estimates the amount of transistor switching. The values of these parameters are partly estimated; however more detail of how they are derived and a good validation of their results is given by Brooks et al. [2000].

Wattch was designed as a tool to explore trade-offs in processor architecture design, and to enable compiler writers to optimise their software. Whereas its original intended use was to allow designers to investigate speed versus power consumption trade-offs in hardware design, here I use it to measure trade-offs between the functional quality and power consumption of software. As a result of the design objectives of Wattch, care has been taken to ensure it is relatively fast to execute hence a program can be evaluated in tens of seconds, an expensive fitness evaluation but viable when used in a parallelised framework.

6.4 GP PARAMETERS

Parameters for the GP search are given in Table 2. The table shows the major parameters of the search, which was implemented using the ECJ 16 Toolkit [ECJ, 2009]. The settings for parameters not listed here are given by the parameter files koza.params, simple.params and ec.params supplied with the toolkit. All of these parameters were taken from Lamenca-Martinez et al. [2006], with the exception of tournament size (since none was given) and the parameters for the multi-objective Strength Pareto Evolutionary Algorithm 2 (SPEA2), since SPEA2 was not used in that paper. The ECJ default was selected for tournament size. The aims of these experiments were to demonstrate the validity and potential of the multi-objective approach to explore functional trade-offs in general, and so no parameter tuning was attempted.

No typing mechanism was employed, and all arguments and return types are handled as unsigned integers. Fitness evaluation was performed in C, by converting the GP tree to a C expression and enclosing it within a function, and then compiled. As some of the members of the function set are not available as native C functions (CSL, CSR), they were implemented as functions within the test harness code and the output of an individual from ECJ was altered to replace the relevant infix expressions with function calls. This has implications for power efficiency, which is discussed further in Section 6.8.2.

6.5 FITNESS MEASUREMENT

The fitness of an individual is determined by its ability to satisfy two objectives: reducing power consumption and optimising functionality or “randomness”. In fact, its fitness is also dependent on the other individuals within the population due to the fitness sharing used by SPEA2, which incorporates the concept of Pareto dominance (see Section 5.6.2 for further details). Within this chapter, “performance” will be used to refer to the functional objective of improving the quality of the PRNG.

6.5.1 *PRNG Quality*

An individual’s performance as a PRNG is measured by the way its output varies when a single input bit is changed. Ideally, when a single bit in the input is flipped, on average half of the output bits should change.

To describe in more detail: to evaluate the performance of the ith individual as a PRNG, a set of 8 random 32-bit integer inputs $a_0, a_1, \ldots, a_7$

Problem Description	Find $p(a_0, a_1, \dots, a_7)$ where p minimises the χ^2 fitness metric defined by the strict avalanche criterion.
Function set	MULT, AND, SUM, NOT, OR, XOR, Logical Shift Left (LSL), LSR, Circular Shift Left (CSL), CSR
Terminal set	$a_0, a_1, \dots, a_7$ and Integer Ephemeral Random Constants (ERCs)
Population Size	150
Generations	250
Crossover Probability	0.8
Reproduction Probability	0.2
Mutation Probability	0.0
SPEA2 Archive Size	100
Selection method	Tournament Selection, size 7

Table 2: GP parameter settings.

is generated by a Mersenne Twister PRNG [Mersenne Twister PRNG, 2009]. The output $p_i(a_0, a_1 \dots, a_7)$ of the individual for these inputs is evaluated, this is also a 32-bit integer. Then one randomly selected bit of one randomly selected integer input is flipped to provide a new set of inputs $b_0, b_1, \dots, b_7$ and $p_i(b_0, b_1, \dots, b_7)$ is evaluated. This data constitutes the result of one test case, and 4096 such test cases are used to completely evaluate one individual.

This fitness component of an individual is then calculated as defined by the Strict Avalanche Criterion (SAC), first introduced by Webster and Tavares [1986] and analysed in detail by Forré [1990]. The criterion measures nonlinearity, specifically the expected distance between outputs given a single bit flip in the input. Each output bit should have a probability of 0.5 of being flipped when a single input bit is changed, in order to maximise the nonlinearity of the PRNG. Hence, the Hamming distance between the two outputs should follow the binomial distribution $B(n, \frac{1}{2})$. By recording the Hamming distance between $p_i(a_0, a_1, \dots, a_7)$ and $p_i(b_0, b_1, \dots, b_7)$ for each test case, a χ^2 squared goodness-of-fit measure can be calculated against the ideal binomial distribution of bit flips. The performance measure of an individual i is given by:

$$\sum_{i=0}^{n} \frac{(C_i - E_i)^2}{E_i} \tag{6.2}$$

Here, n is the number of possible bit flips $(0, 1, \dots, 32)$, C_i is the observed frequency of test cases that resulted in i bit flips, i.e. where the Hamming distance was i between $p_i(a_0, a_1, \dots, a_7)$ and $p_i(b_0, b_1, \dots, b_7)$, and E_i is the expected frequency of tests with i bit flips according to the binomial distribution. Our aim is to minimise this statistic, and reduce the deviation from the ideal distribution.

Note that in order to test an evolved PRNG, I employ a Mersenne Twister PRNG! This circularity does not, however, pose a problem for fitness evaluation. The only danger would be if an evolved PRNG managed to "take advantage" of some predictable characteristics of the input integers, generated by the Mersenne Twister. Given that the Mersenne Twister algorithm is designed not to betray such characteristics, this appeared to be an unlikely outcome!

6.5.2 *Employing Multi-objective Optimisation*

This is a bi-objective problem and hence I use Genetic Programming combined with a suitable MOO method, SPEA2. This enables me to examine a wide range of possible solutions, as opposed to a single one located by a weighted sum method, or a subset found using other techniques. Discovering a range of trade-offs is important here because we wish to answer questions about the impact of design decisions on the objective space as a whole. As noted in Section 1.2 such questions can only be answered in a relative sense.

An implementation of the SPEA2 algorithm was written as an extension to ECJ, which followed precisely the original algorithm specified by Zitzler et al. [2001]. For details of SPEA2, see Section 5.6.2. The archiving is effectively elitist, and counteracts the emergence of bloat in GP, because a larger individual will only survive if it makes an improvement over the existing archive in at least one objective. This is an important feature in attempting to evolve low power individuals, because unchecked bloat would increase the number of instructions in individual solutions. In the case where those instructions are ineffective yet executed code (for example, identity function execution), this would result in increased power consumption.

6.6 PROBLEM SUMMARY

To summarise, the problem to be solved was to find a Pareto front of programs, where each program implemented a function $p(a_0, a_1 \dots, a_7)$. The two objectives, both of which were cost functions to be minimised, were the power consumption of the program and the measure of randomness given by the χ^2 goodness-of-fit measure.

6.7 IMPLEMENTATION

Figure 16 gives an overview of the way that the ECJ-based framework measures the fitness of a program. The PRNG is a symbolic expression, and is written out to a file as a C function. This file is then cross-compiled for the MIPS architecture (on a Linux host) and linked with test harness code, which measures the fitness of a program using the χ^2 measure. The test cases are generated using a Mersenne Twister PRNG. The program is then run on the Sim-Wattch simulator, which produces power statistics along with redirected program output from the test harness i.e. the χ^2 measure. These fitness values are used by SPEA2 to assign fitness scores based on Pareto non-dominance and a niching function.

An alternative arrangement considered was to pass test case information to and from the simulator via streams or sockets, but this proved

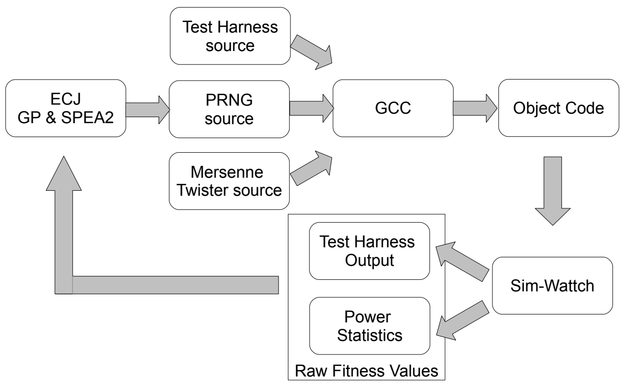


Figure 16: An overview of fitness evaluation.

too inefficient to be feasible. By placing the entire fitness evaluation (test case creation, the program under test and the goodness-of-fit calculation) within the simulator, the run-time of a single program evaluation was greatly reduced. As a further improvement, the expected distribution for a given number of test cases was calculated once and hard-coded in the test source. However, even with this efficient arrangement, a single evaluation of an individual over 4096 test cases took approximately half a minute on a 4200+ AMD processor. Most of this time was spent within the Sim-Wattch simulator, hence the only remaining target for optimisation was to reduce the sample size.

6.7.1 *Reducing Sample Size*

The paper that this work builds upon [Lamenca-Martinez et al., 2006] used 16384 test cases, whereas in this work I chose to use only 4096 test cases. This section justifies that decision.

To reduce the number of test cases required, I evaluated the variance of functional fitness measures on smaller sample sizes using a bootstrapping resampling technique [Berthold and Hand, 1999]. Firstly, I executed a single run using the larger test sample size over 4 days and logged each individual in the archive. A selection of 5 individuals were chosen across a range of different fitness values to provide the required data. These individuals were evaluated and the $p(a_0, a_1, \ldots, a_7)$ and $p(b_0, b_1, \ldots, b_7)$ values logged over 16384 test cases. This allowed me to employ statistical bootstrapping methods to determine whether smaller sample sizes were effective in estimating the χ^2 measure.

An example plot is given in Figure 17. This illustrates for a single program the impact of varying the sample size on the resulting fitness measure. The sample sizes run from 0 to 16384 in powers of 2. Each point is the result of using bootstrapping with 30 bootstrap samples. Other results had similar or smaller variance, and from these plots I concluded a sample size of 4096 was sufficient. It was necessary to ensure that the smaller sample size did not adversely affect power consumption statistics, and similarly the variance of the power statistics for each program was empirically found to be very low at a sample size of 4096. Note that during the experiments each individual was also re-evaluated at each generation, reducing the chance of a single outlier

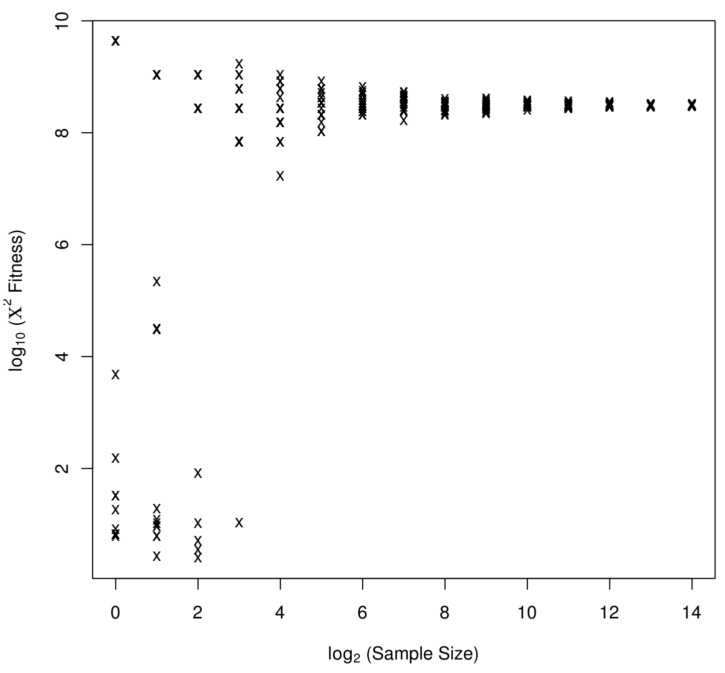


Figure 17: Example plot of sample size against fitness for one program.

fitness measure incorrectly giving an individual a higher priority when populating the next archive.

6.8 RESULTS

6.8.1 *Example Pareto Fronts*

Figure 18 shows the archive at generation 249 of Experiment 1, where each point corresponds to a program's properties in objective space. The power consumption is the total power consumed by each individual across all 4096 test cases, and includes that consumed by the test harness. Fitness is the goodness-of-fit measure as described in Section 6.5.1. Note that the archive is not always composed entirely of non-dominated individuals for two reasons: firstly it may be "topped up" with dominated ones by the SPEA2 algorithm when there are too few non-dominated individuals to fill the archive completely, and also due to the variance in fitness values (which are input-dependent) when the archive is re-evaluated at each generation.

This graph demonstrates that functional trade-offs are indeed possible for this problem. The most impressive PRNGs, at the bottom-right corner of the diagram, have a small deviation from the binomial distribution, and these require the most power. Very poor pseudorandom number generators, with lower power requirements, are at the top-left of the diagram. This diagram allows us to visualise the relationship between power consumption and functionality of the problem.

There is a discontinuity in the archive, where a step improvement in the nonlinearity of the evolved PRNGs is observed. This is caused by

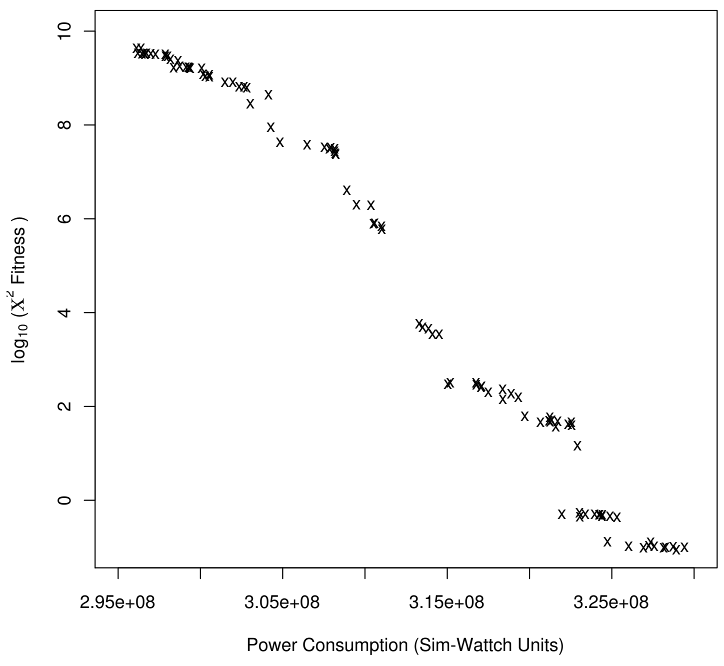


Figure 18: Archive at Generation 249, Experiment 1. The graph shows the trade-offs made by programs within the archive, between total power consumption and error. For both objectives, lower values are better.

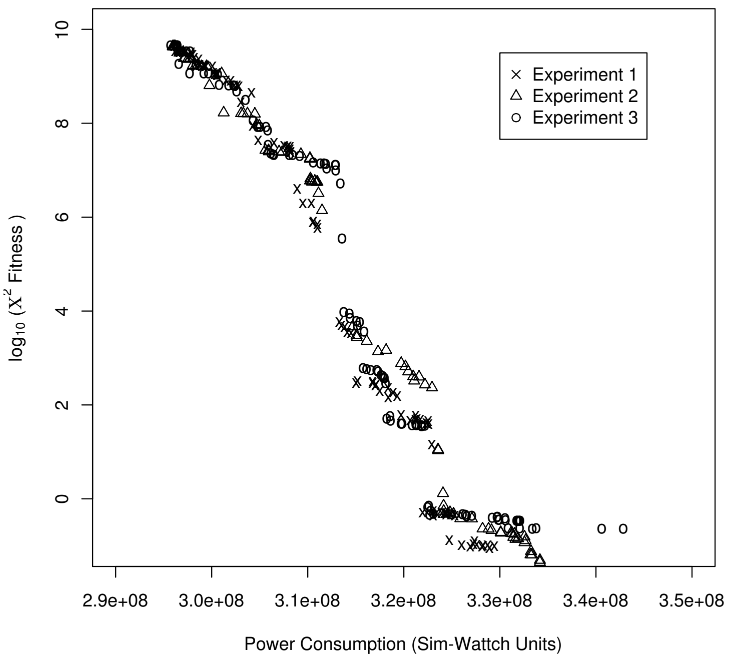


Figure 19: Archives at Generation 249 for Experiments 1, 2 and 3. Similar trade-offs are discovered.

the simple niching function used by SPEA2 algorithm, which works on Euclidean distance between points in objective space. As the fitness values are not normalised, from this point to the right of the graph the power consumption, rather than the PRNG performance, dominates the niching function. This will have some impact on the variety of solutions produced, depending on how often the niching function is used by the search. The use of a more sophisticated niching function would enable further control on how the archive approximates the Pareto front.

Figure 19 shows three archives resulting from three separate experiments, with different seeds for ECJ being the only difference between each experiment. Similar trade-offs are achieved, although Experiment 2 shows better performance in terms of optimising functional fitness.

These results are an order of magnitude worse, in terms of the functional fitness (SAC) than the results presented by Lamenca-Martinez et al. [2006]. The aim of this work is not to improve on those results, however Experiment 2 was extended for an extra 50 generations (i.e. fifty further generations after the archive in Figure 19 had been produced) to demonstrate that the quality of solutions found was not compromised by employing MOO. The Pareto front was improved by a small amount and χ^2 values as low as 0.00195 were obtained[1]. The p value for this result, effectively giving the probability that this sample is drawn from the ideal Binomial distribution, is 1.00 (to 3.s.f). This is

1 Please note that the results reported in Lamenca-Martinez et al. [2006] and White et al. [2008] are based on counts rather than frequencies: to compare those values to these results it is necessary to divide by the sample size used. For example, Lamenca *et al.* report a value of 12.7 over one set 16384 samples, giving a χ^2 value of 0.00775

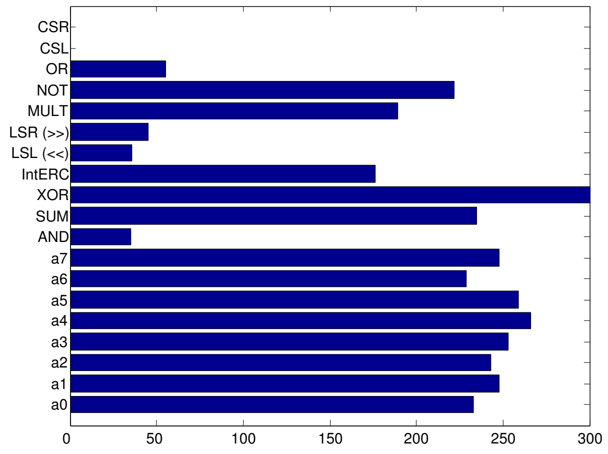


Figure 20: Function set usage across the archives in Generation 249 for Experiments 1, 2 and 3.

an excellent result: see Figure 21 for an indication of the quality of the best individual from the extended run of Experiment 2.

6.8.2 *Function Usage*

Figure 20 shows the number of programs using each function in the function set for the 300 individuals contained in the final archives from Experiments 1, 2, and 3. For each function, the count represents the number of individuals that the function was used within. This diagram gives an intuitive impression of how important each function within the function set was for developing useful trade-offs. The figure begins to answer the question of whether the MULT function should be included in the function set: at this stage it does indeed appear to be a useful tool in making trade-offs, as it is used by most of the programs on the Pareto front. This question is addressed in more detail in Section 6.8.4.

From the chart, the most striking feature is the lack of Circular Shift Left (CSL) and Circular Shift Right (CSR) function calls. As C does not provide these operators, they were included within the test harness as functions (in the same way as Lamenca-Martinez et al. [2006]). The function calls are expensive, both because of the overhead in calling a function, and because the CSL and CSR functions required several lines of code. Hence they were discarded by the search. However, instruction sets for processors such as the Z80 do provide such rotate instructions. By evolving programs in assembly language, the search could take advantage of these commands. This raises an important issue of how the target language impacts the ability of search to make trade-offs.

6.8.3 *Example Individuals*

Three example individuals are shown in Table 3. These individuals are the best (in terms of functional SAC) from the extended run of Experiment 2, and the worst and median from Experiment 1. The χ^2 measure of the best individual is the average of 10 runs over 16384 test cases, as the variance in the fitness function becomes significant at very low χ^2 values.

Relative PRNG Functional Performance	Program Expression	Power Consumption (Wattch Units)	χ^2 Fitness
Best (Experiment 2)	$(2307363674 \oplus (a2 * a6)) + ((\neg((((((a2 * a6) \oplus (a7 \oplus a1)) * (a0 \oplus a3)) \oplus (a2 * a6)) + (a5 * a4)) >> 2307363674) \oplus (a0 \oplus a3)) + \neg((a5 * a4) * ((2307363674 \oplus (a7 \oplus a1)) + (a0 \oplus a3))))$	$3.3658 * 10^8$	0.00195
Worst (Experiment 1)	$\neg(\neg(1997453768))$	$2.9639 * 10^8$	$1.07 * 10^9$
Median (Experiment 1)	$a2 \vee \neg(((a2 + a0) * ((a4 \oplus ((a6 + a5) \oplus a7)) + (a1 \oplus a3))) \wedge \neg(\neg(1997453768)))$	$3.2892 * 10^8$	$1.46 * 10^5$

Table 3: Three example PRNGs of varying functional quality.

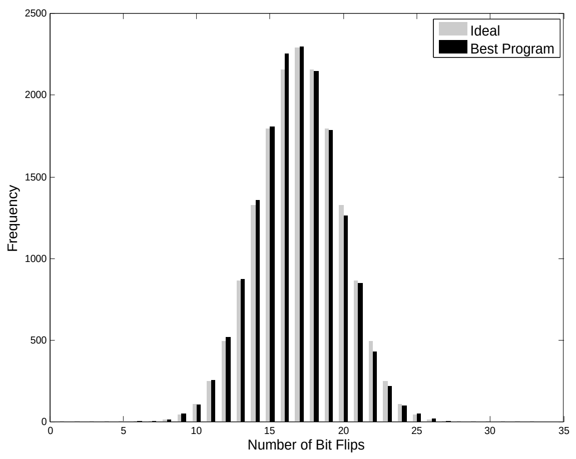


Figure 21: Distribution of bit flips of the best individual from Experiment 2.

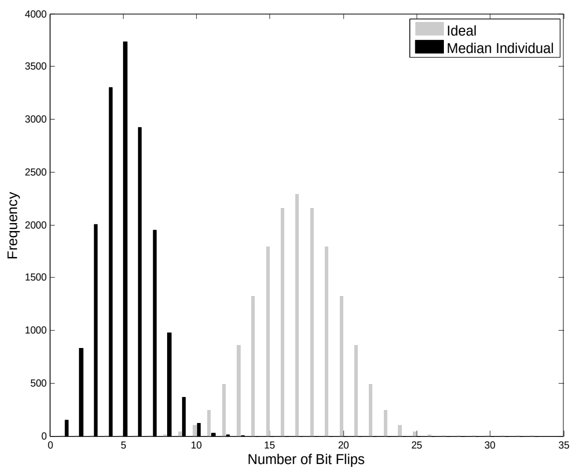


Figure 22: Distribution of bit flips of the median (by χ^2 Fitness) individual from Experiment 2.

Test	Best Individual
Entropy	7.999999 bits/byte
Compression Rate	0%
χ^2 Statistic	264.98 (32%)
Arithmetic Mean	127.5011
Monte Carlo π Estimation	3.141828142 (0.01%)
Serial Correlation Coefficient	0.000010

Table 4: ENT results for the best individual.

The distribution of the bit flips from test cases for the best individual from Experiment 2 (extended to 300 runs) is given in Figure 21. Comparing this to the median individual of Experiment 1, which is given in Figure 22, these diagrams illustrate the dependence of quality on power, or "bang for your buck" as far as random number generation is concerned.

It is interesting to note that the best individual in Table 3 contains repeated components, and also that the use of the MULT function is distributed across different parts of the program tree. This would appear to be a sensible way of managing the "energy budget" of an individual through placement of the most expensive function.

The best individual was then used to generated a 250MB file of random bytes, by initialising $a_0, a_1, \ldots, a_7$ with random numbers and from then onwards feeding the previous output into the next input at position 7, as described by Lamenca-Martinez et al. [2006]:

$$a^{n+1} = a^n_{i+1} \; \forall i = 0, 1, \ldots, 6 \tag{6.3}$$

$$a^{n+1}_7 = p^n(a_0, a_1, \ldots, a_7) \tag{6.4}$$

The file was then run through the ENT test suite [ENT, 2009], a standard battery test used to evaluate PRNGs. The results are given in Table 4. These results indicate that with this feedback method the individual performs well as a PRNG. However, it performs poorly given a low entropy input such as feeding sequential numbers, particularly on the ENT χ^2 test (not to be confused with the avalanche criterion test used in this chapter!). It is not surprising that this is the case, because fitness evaluation was based on tests using random rather than sequential inputs. More surprisingly, the best individual "Lamar" as reported by Lamenca-Martinez et al. [2006] performs well under a low entropy input. It is possible that a comparable number of experiments would have to be completed to achieve comparable results, in particular to achieve a similarly low value for the ENT χ^2 objective. An alternative method may be to incorporate low entropy tests into fitness evaluation.

6.8.4 *Impact of the MULT Function*

In previous work [Hernandez et al., 2004, Lamenca-Martinez et al., 2006], there was some discussion as to whether the MULT function should be included in the function set, as whilst it is likely to aid greatly in achieving the functional goals of pseudorandom number generation, it is an expensive operation. Using a systematic approach based on

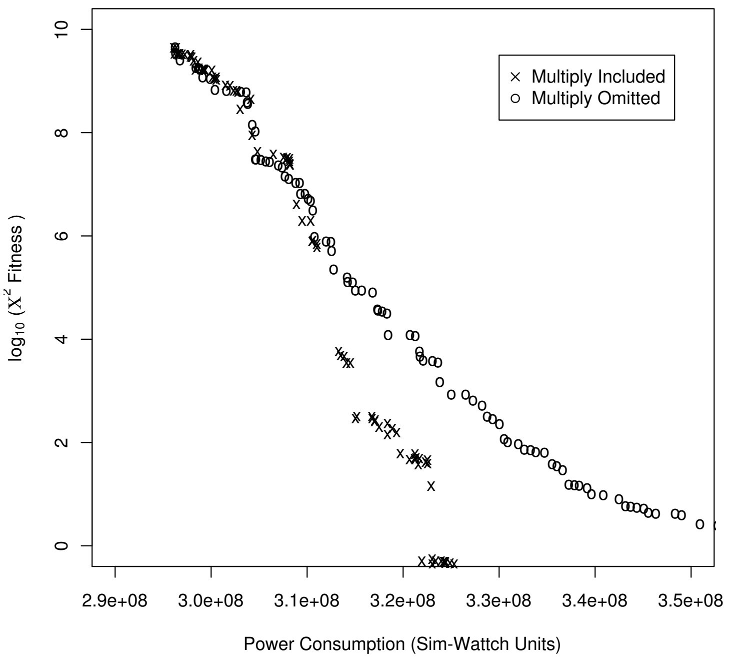


Figure 23: Pareto front from Experiment 1 compared to Pareto front without multiply function.

multi-objective optimisation, we can address this issue more directly, by comparing the Pareto fronts that result under experiments using (a) the function set described above and (b) the same function set with the MULT function removed.

Figure 23 displays representative examples of both Pareto fronts, and the difference is quite striking. Without the MULT function, the Pareto front is stretched to the right, in that the same level of functionality requires an increased amount of power. Furthermore, certain levels of functionality do not seem achievable without it, at least given the computational resources provided to the search.

6.9 SUMMARY

This work demonstrates that GP can be used in combination with MOO as an effective method for exploring the trade-offs between power consumption and functionality when creating a pseudorandom number generator. It also demonstrates that continuous trade-offs are possible for the PISA-based SimpleScalar architecture for this problem. The idea that such a granularity of trade-off might exist was previously non-obvious, but it is evidently so and we may expect to find such trade-offs across a variety of domains. This could enable the dynamic management of a system where an entire set of software components could operate at different levels of efficiency and functionality, much like dynamic power management (see Section 3.3.5). A designer may alternatively choose a satisfactory compromise in a static design from the set provided.

Given the general nature of the PISA architecture, and that only relative measurements were required, it is likely that solutions produced in this experimentation will also work well on other platforms, and that the approach in general should transfer well.

To achieve the same level of functionality as results obtained in previous work, when using MOO, requires increased computation power. I extended the search from 250 generations to 300, i.e. an increase of 20% in terms of evaluations, genetic operators etc.

My approach gives a systematic way of answering questions about the importance of individual operations in achieving a non-functional requirement. For this specific problem, the MULT operator is a useful function to include in the function set. Whilst the power consumption of any instruction varies depending on the architecture, it is likely that this conclusion will generalise to other architectures. The decision over whether MULT is necessary is reminiscent of the problem of instruction subset selection discussed in Section 3.3.1. We could potentially carry out similar analysis to select instructions we require and subsequently specify those functions for implementation on a programmable hardware platform.

This work acts as a proof-of-concept for similar applications in creating hash functions or other embedded system software components. Equally, a great number and variety of non-functional requirements can be considered.

In this chapter, I have created resource-efficient software from scratch, producing a software artefact that could be deployed immediately, for non-cryptographic applications at least. The next step in my investigation is to see whether this success can be repeated in the task of improving existing software: can GP be used to *increase* efficiency as well as to create it?

7 IMPROVING RESOURCE EFFICIENCY

7.1 INTRODUCTION

The previous chapter demonstrated the application of Genetic Programming, multi-objective optimisation and simulation in order to produce entirely new software (pseudorandom number generators) with low-power consumption. In this chapter, I demonstrate a way in which *existing programs* can be improved to have desirable non-functional properties. Execution time is the property of interest here, but the method generalises to other requirements.

In this chapter, it is assumed that a manually written solution to a problem exists, and this code is then taken as input to a system that attempts to improve it with respect to its execution time (estimated by the number of instructions executed). As discussed in Chapter 3, compilers already attempt to perform a limited amount of such optimisation. However, in general they cannot restructure a program's implementation without restriction, and even when using a limited set of semantics-preserving transformations they cannot always find the most effective sequence to apply. Compilers focus on localised optimisations and we cannot expect a compiler to eliminate unnecessary loop iterations, or shortcut base cases in a recursive function.

In optimising software, this chapter considers its operational profile [Beizer, 1990], i.e. its expected input distribution, allowing the Genetic Programming search to exploit the anticipated usage of software. The original manually-written code is exploited both to seed the initial population and as an oracle for testing purposes.

Please note that this work was carried out in collaboration with Andrea Arcuri from the University of Birmingham, some material has been published [Arcuri et al., 2008] and a journal extension submitted for review. Most of the design work was done together, and effort for implementing the code of the actual framework was evenly divided. The bulk of the analysis reported here was carried out by myself.

7.2 PROBLEM FORMULATION

The problem to be solved may be formulated as follows: given an existing program function as input, p_0, and an expected distribution over input values, find an improved function p^* such that:

$$\begin{aligned} f_e(p^*) = 0 &\quad \text{for the functional objective } e \\ f_l(p^*) &\quad \text{is minimised for each non-functional} \\ &\quad \text{objective } l \end{aligned} \tag{7.1}$$

where achieving $f_e(p^*) = 0$ approximates semantic equivalence between p^* and p_0 with respect to a test set T. In this chapter, only two objectives are considered: the functional objective f_e and the non-functional objective f_{inst}, the number of instructions executed over a set of test cases. This work could be extended to satisfy multiple non-functional objectives.

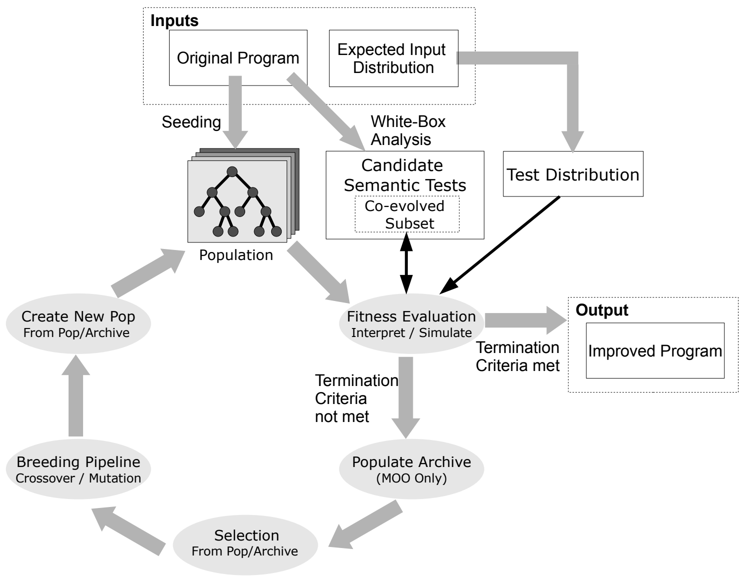

Inputs
Original Program
Expected Input Distribution
Seeding
White-Box Analysis
Population
Candidate Semantic Tests
Co-evolved Subset
Test Distribution
Create New Pop From Pop/Archive
Fitness Evaluation Interpret / Simulate
Termination Criteria met
Output
Improved Program
Termination Criteria not met
Breeding Pipeline Crossover / Mutation
Populate Archive (MOO Only)
Selection From Pop/Archive

Figure 24: Evolutionary framework.

7.3 TARGET PLATFORM

The execution time of a program is dependent on the host hardware environment, and I have again used simulation to estimate the execution time of individuals. Specifically, the M5 [Binkert et al., 2006] simulator was used, targeted for an Alpha RISC ISA based on a DEC Tsunami system. The simulator was used at a course-grained level of detail, such that cache activity and pipelining details were ignored. This enabled faster evaluation suitable for large-scale experimentation, and also is likely to improve the generality of any optimisation. The limitation is that optimisations cannot be made by manipulating very fine-grained timing details (more on this in Chapter 8).

7.4 PROPOSED FRAMEWORK

The prototypical framework used to solve the problem outlined above is given in Figure 24. The framework is a development of that given in Figure 16 used in Chapter 7. It has been extended as follows:

- An original program and expected distribution of inputs are now given to the framework.
- Two separate sets of test cases are used, and coevolution of test cases is employed.
- The non-functional property (instruction count) is evaluated through modelling in addition to simulation.
- Strongly Typed GP is used.

The framework takes as input the code of a function, expressed in the C programming language, along with an expected input distribution. It attempts to reduce the program's instruction count on the Alpha-based target platform. The general nature of the approach means it can easily be applied to other target languages and platforms. It applies GP to optimise one or more non-functional criteria, whilst ultimately attempting to maintain semantic equivalence with the original program. This framework is a prototype, and part of the purpose of the experimentation in this chapter is to assess whether the proposed use of MOO and coevolution in particular are beneficial, in terms of their impact on the ability of the framework to optimise non-functional properties of the software.

Strongly Typed GP is used here to construct a search space expressive enough to encompass the original solution for each of the case studies considered. See Section 4.9.1 for more on Strongly Typed GP. The return and argument types of all functions are given in Table 5, omitting problem-specific variables. From this table, the complexity of the type system becomes apparent. Note that a node may have more than one return type: I am using both atomic and set-based typing. For example, an increment operator ++ may be a standalone statement or it may be part of a `for` loop. An example symbolic tree, equivalent to the case study Sort1, is given in Figure 25. "RA" stands for "Read Array" and "WV" for "Write Variable". The code for Sort1 is given in Figure 32.

The framework is quite unique amongst GP experimentation in the way that the first generation is initialised (seeded), how the test sets are created, and the particular use of MOO.

7.4.1 *Seeding*

Rather than attempt to create equivalent software from scratch [Reformat et al., 2007] seeding can be used to exploit the original solution p_0. GP systems predominantly create an initial population using Koza's established ramped half-and-half method [Koza, 1992] as described in Section 4.4, but here three alternatives based on the concept of using the input program as a starting point are tested. The intention is that the genetic material of the input program can be used as *building blocks* in evolving an improved solution. In terms of GP schema theory, it is intended that good solutions may be found by exploring schemata containing or somehow "close to" the original solution.

In designing seeding strategies we face a classic exploration versus exploitation trade-off that is so often an issue in heuristic search, and in particular evolutionary computation. On one hand, over-exploitation of the original program might constrain the search to a particular suboptimal area of the search space, i.e. the resulting programs will be very similar to the input one. On the other hand, ignoring the input genetic material would potentially make the search too difficult.

There are further dangers, as discussed in Poli et al. [2008]: including many highly fit individuals along with those generated by other means may lead to a lack of diversity in the following generations, and including too few fit individuals may result in the loss of this initial guidance. Therefore the seeding strategy is the subject of experimental investigation, and the following types of seeding are tested:

- *Standard*: Koza's ramped half-and-half initialisation method. See Section 4.4.

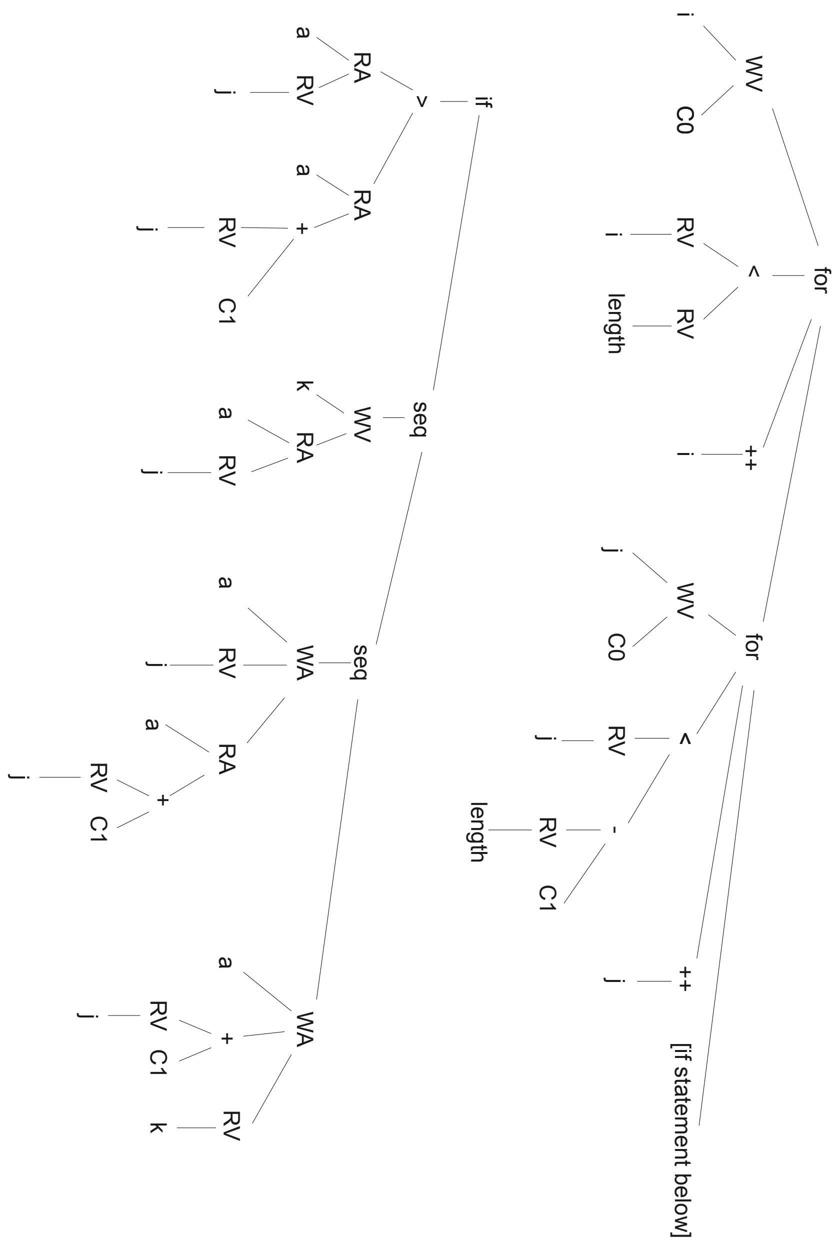


Figure 25: Tree representation of Sort1.

Function	Return Type	Children
$0, 1, \ldots, 10$	int	n/a
$+, -, *, /, \%$	int	int, int
$--, ++$	increment, statement	variable
$==, >, \geq, <, \leq$	Boolean	int, int
&&, ||	Boolean	Boolean, Boolean
!	Boolean	Boolean
true, false	Boolean	n/a
V_tmp	Variable	n/a
ReadVariable	int	Variable
WriteVariable	Statement, Assignment	Variable, int
VariableWrapper	Variable	Variable
ReadArray	int	ArrayVariable, int
WriteArray	Statement	ArrayVariable, int, int
ArrayWrapper	ArrayVariable	ArrayVariable
skip	Statement, Increment, Assignment	n/a
return	Statement	int
while	Statement	Boolean, Statement
if	Statement	Boolean, Statement, Statement
Sequence	Statement	Statement, Statement
for	Statement	Assignment, Boolean, Increment, Statement
switch	Statement	int, case, statement
case	Statement	int, Statement
case_sequence	Case	Case, Case
inline if	int	Boolean, int, int
fac	Statement	int

Table 5: Function types for all experiments.

- *Cloning*: a fraction of the initial population will be replaced by a copy of the input function. The remaining individuals are then generated using the *standard* method.
- *Delta*: a fraction of the initial population will be created by making a copy of the input program and by applying a random mutation to each individual using one of the mutation methods employed during the rest of the evolutionary run. A mutation can be construed as a "step" away from the original program in the search space and creation of an individual that belongs to some of the same schemata as the original. The remaining individuals are generated using the *standard* initialisation method.
- *Sub-tree*: a fraction of the initial population will be composed of copies of randomly selected subtrees taken from the original program. The selected subtrees must have a root node type-compatible with the root node of the input program, such that they return the correct type. The remaining individuals are generated using the *standard* initialisation method. This method would be most effective if the building block hypothesis applies to GP: it would then facilitate the recombination of these subtrees to improve efficiency (see Section 4.8.1).

Since the first paper on this work was published, Schmidt and Lipson [2009] have also investigated similar seeding methods as a way to incorporate expert knowledge into a search process.

7.4.2 *Preserving Semantic Equivalence*

Maintaining semantic equivalence is the biggest challenge facing any attempt to solve the problem outlined in Section 7.2. In this work, semantic equivalence of the output program cannot be guaranteed. The proposed technique is probably of more use as a method of gaining insight into potential optimisations rather than a fully automated approach at this stage. For example, the output from the framework could be verified by manual inspection or using data-mining methods. However, the raw optimised output function may be immediately useful in applications that do not have a Boolean measure of acceptable functionality – such as the application discussed in Chapter 6.

In this chapter, I recast the problem of software optimisation. Rather than limiting optimisation to semantics-preserving operations only, and then attempting to find the best such operations in order to achieve maximum improvement of a quality metric, a dramatically different approach is followed. The framework will try to optimise the software for the quality metric with arbitrary transformations, which is easily done (e.g., if it is execution time then we can simply remove an instruction), and turn the problem into one of mutation testing [Andrews et al., 2006]. Can we tell the difference between the original and the improved program? If we fail to find test cases that can discern between the original program and the optimised one, despite a great deal of effort, we have a good degree of confidence that we have found a semantically equivalent yet optimised version. The latter may be described as "Search and Filter" optimisation. This is illustrated in Figure 26.

Within the framework, the "filter" employed is the use of coevolved test cases, as well as a static set that guarantees branch coverage over the

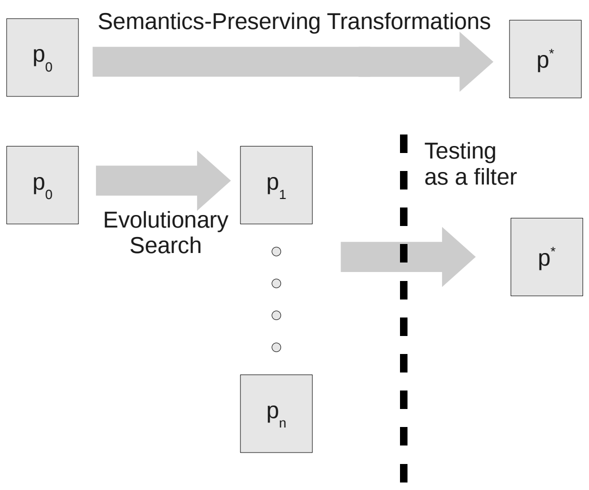


Figure 26: Optimising software by search and filter, rather than semantics-preserving transformations.

original code. The use of coevolution prevents overfitting, whereby the search could potentially exploit a fixed set of test cases to improve performance at the expense of removing semantic equivalence on untested areas of the input domain. The method employed is in principle similar to that used in seminal work by Hillis [1990].

Before the evolutionary algorithm begins, a large set of test cases is generated using a white box testing criterion [Myers, 1979], specifically branch coverage. This set is partitioned into subsets, one for each branch of an input program, which ensures a degree of behavioural diversity amongst test cases. This initial set of test cases could be generated using any existing automated testing method [McMinn, 2004, King, 1976].

The test set is then coevolved as a separate population (the "training set"), a changing subset of the larger pool produced prior to evolution. This training set is also partitioned, such that it ensures branch coverage of the original program and therefore that the test set at any given generation encapsulates the semantic notions encoded in those branches. Figure 27 illustrates the relationships between a program and the corresponding test set populations.

Note that only test cases with valid inputs are generated. Inputs for which the pre-condition of the program is not satisfied are not generated, because *any* corresponding output would be valid. For example, for functions with arguments including a pointer to an array and a variable representing its length, only valid values for the length variable are generated.

At each generation, the GP individuals are tested with the test cases in the training set. The sum of the errors from the expected results is referred to as the *semantic score* and is one component of the fitness of a GP individual. *These semantic scores are also used to define the fitness value of the test cases*. Each program tries to minimise the semantic score on each test case, whereas each test case tries to maximise this score on each program.

For each subset of test cases only the best half (based on their fitness values) is retained at each generation. The other half is randomly replaced by test cases in the large pool produced prior to evolution. These

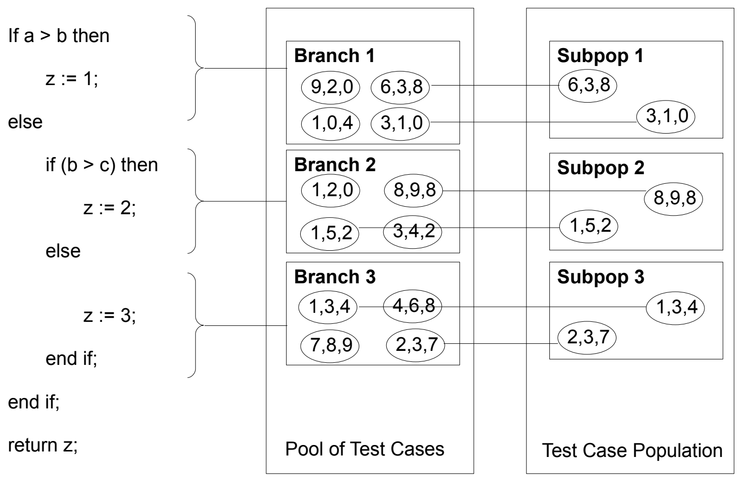


Figure 27: The relationship between an input program and the semantic test set population.

replacements are performed such that each test case in the training set is unique. The best individual from each subset is stored in a *Hall of Fame* archive [Rosin and Belew, 1997]. The fitness of individuals is also based on their execution on test cases from this archive.

When coevolution is not employed, the test cases are chosen stochastically at each generation. In this situation, they are chosen based on the operational profile, in the same manner as test cases used to evaluate non-functional properties.

7.4.3 *Evaluating Non-functional Criteria*

In evaluating non-functional criteria, a separate training set from that used to evaluate the semantic score is employed. The set is drawn from the expected input distribution provided to the framework, which could (for example) be based on probe measurement of software.

The set of non-functional tests is resampled from the expected input distribution at each generation, to prevent overfitting of non-functional fitness for a particular set of inputs. The final fitness of a potential solution (a GP individual) will be composed of both its semantic score and this measurement of its non-functional property.

We must be able to reliably (in a repeatable and relatively accurate manner) estimate or measure the non-functional property concerned, and in this case the total number of instructions executed is the chosen metric. This is preferred over the cycle count as a reliable high-level estimate of execution time. It is computationally less expensive to compute than more detailed timing information.

In this chapter I use both simulation, in a similar manner to Chapter 6, and also modelling of instruction count. The latter is motivated by the computational expense of simulation.

SIMULATION The instruction count of an individual can be estimated using a processor simulator and the M5 Simulator [Binkert et al., 2006] is used, targeted for an Alpha Microprocessor. The parameters of the simulator describing the processor and surrounding architecture were left unchanged from their default values, although a few small code changes for convenience and efficiency were applied. The choice of this particular target ISA was motivated by the goals of replicability, efficiency and accuracy. The free availability of the simulator allows others to replicate this work, and its source code and implicit processor model are available for anyone to review. M5 is a modern platform with great flexibility and is well-supported. The Alpha ISA is the most mature target platform of the simulator.

When employing simulation, individuals are written out by the framework as C code and compiled with an Alpha-targeted GCC cross-compiler. A single function is linked with test code that executes the given test cases, and a total instruction usage estimate provided by parsing a trace file.

Whilst simulation does not perfectly reflect a physical system, it is worth noting that we are again only concerned with *relative accuracy* between individuals, i.e. we wish to *improve* efficiency rather than precisely determine it [Jacome and Ramachandran, 2006]. Incorrect relative evaluation of two individuals will add noise to the fitness function. The difficulties and intricacies accurately simulating complex hardware platforms is an issue beyond the scope of this work: alternatives or improvements in both the simulator and compiler can easily be incorporated into the framework.

MODEL CONSTRUCTION In order to carry out large scale experimentation efficiently, modelling was considered as an alternative to simulation of instruction usage. The instruction count of a program was estimated as a linear model of the frequencies of high-level primitive execution:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots \beta_n x_n + \epsilon \tag{7.2}$$

where Y is the estimated instruction count of a program, $x_1, x_2, \dots, x_n$ are the frequencies with which each of the n functions in the function set are evaluated within a program, and the coefficients $\beta_1, \beta_2, \dots, \beta_n$ are an estimate of the cost of each function. The intercept is given by β_0 and ϵ is the noise term, introduced by factors not considered by the other components of the model.

This model is a large simplification, because compiler optimisations will be dependent on the program structure and there is not a simple mapping between high level source code and low level instruction execution. Simulation provides dynamic information that cannot be captured in such a simple model. The ordering of instructions, for example, is not taken into account. Simulation can be carried out at variable levels of fidelity, where improved accuracy can be provided at the expense of greater computation time.

To use such a model, the coefficients must be estimated. This is achieved by executing one large evolutionary run of the framework for each case study. Each program evaluated during the run is both interpreted in Java and executed through the simulator. From the interpreter, the frequencies with which each high-level primitive (`if`, `while`, array access etc.) is evaluated are recorded, and the corresponding instruction count of an individual as measured by the simulator is logged. Least Squares Linear Regression is then used to fit this model.

It is possible to verify the relative accuracy of this model for the data points used in constructing it, as detailed in the results in Section 7.8.3. We can effectively estimate the impact the model has on decision making in the algorithm as opposed to using simulation.

7.4.4 *Multi-objective Optimisation*

The problem contains two objectives and I use both weighted sum and Pareto-based MOO. The two objectives are:

1. Minimise error across test cases.
2. Minimise the number of instructions executed.

We are concerned primarily with minimisation of error, and secondly with reducing instruction count. Ultimately, we are not concerned with individuals that do not meet their functional specification as estimated by the test set, and so it is sensible to combine our objectives with a weighted sum method, as described in Section 5.4.1. Note that this is a feature of the functions chosen as case studies, which have Boolean values of functional acceptability. By normalising the fitness values and setting the weighting of the functional fitness to be greater than the weighting given to the instruction count, we can create an arbitrarily sized bias towards favouring improvement in functional performance over the instruction count of a solution.

Here, I also attempt to use Pareto-based MOO methods in an unusual application. The difference between the goal of this work against a traditional Pareto-based method is illustrated by Figure 28. Given the input program at the bottom-right of the figure, we are ultimately interested only in locating the desired output program that lies on the x axis as close to the origin as possible.

However, I still employ Pareto-based MOO within this work: this is because I am interested in the balance between exploration and exploitation. A Pareto-based MOO approach explicitly forces the search to discover and retain solutions in Figure 28 that are *not on the x axis*, that is those solutions we may actually regard as *inferior*. The aim is to identify if employing Pareto-based MOO allows the search to locate smaller, instruction-efficient subtrees of a program able to satisfy a subset of test cases. It is hope that these may be reassembled by the search algorithm into improved solutions later in the evolutionary run. This proposal and its effectiveness are linked to the building block hypothesis and GP Schema theory: see Section 4.8.1 and Langdon and Poli [2002].

In summary, two approaches combining objectives in fitness evaluation were used. The first therefore is a simple linear combination of the functional and non-functional fitness measures. The second is the Strength Pareto Evolutionary Algorithm Two (SPEA2) as outlined in Section 5.6.2 and used in Chapter 6. The choice between the two is the subject of experimental evaluation in Section 7.7.1.

Any advantage found by using a Pareto-based method has interesting implications for understanding how GP achieves its goal: *can building blocks be recombined in different ways to improve performance?*

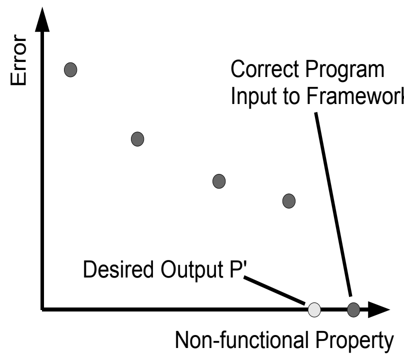


Figure 28: A Pareto front composed of five programs in objective space.

7.4.5 *Fitness Function*

Given a population of programs P, two sets of test cases T_0 and T_1 are used. The fitness function of the programs in P is based on their execution on these test case sets. The set T_0 is used to evaluate the semantic score of the programs and undergoes coevolution. The set T_1 is used to assess the non-functional value of the programs, i.e. the instruction count, and is a sample drawn from the expected input distribution.

Given $f_e(p, t_0)$ a function to calculate the semantic score (error) of a program $p \in P$ run on a test $t_0 \in T_0$, and $f_{inst}(p, t_1)$ its instruction count on a test $t_1 \in T_1$, the fitness function $F(p, T_0, T_1)$ to be minimised is as follows:

$$F(p, T_0, T_1) = \alpha \cdot \text{norm}\Big(\sum_{t_0 \in T_0} f_e(p, t_0)\Big) + \beta \cdot \text{norm}\Big(\sum_{t_1 \in T_1} f_{inst}(p, t_1)\Big) \tag{7.3}$$

where norm is any normalising function to the range $[0, 1]$; here Koza's adjusted fitness (as in Equation 4.2) was used. The priority between objectives is determined by the weightings α and β. The values $\alpha = 128$ and $\beta = 1$ were chosen, giving more emphasis on the semantic score as used in Luke and Panait [2006]. In the cases in which a program is either not compilable or it has runtime errors (see Section 7.6), then a death penalty is applied, i.e. its fitness value is $\alpha + \beta$. When Pareto-based MOO is used, these two objectives (normalised semantic score and instruction score) are treated separately.

When employing coevolution, each individual test case in T_0 must be assigned a fitness, f_{tc}, the sum of all program fitnesses measured on that test case:

$$f_{tc}(t_0, P) = \sum_{p \in P} f_e(p, t_0) \tag{7.4}$$

In the case of array outputs, the fitness of an individual depends on the state of the memory after the computation. After evaluation each value in the modified array A' is compared against the value in the

Name	LOC	GP Nodes	Input	LV
Triangle1	35	107	int , int , int	1
Triangle2	38	175	int , int , int	1
Sort1	11	63	int[] , int	3
Sort2	18	69	int[] , int	4
Factorial	7	16	int	0
Remainder	40	208	int , int	3
Swi10	22	68	int	1
Select	94	392	int[] , int , int	9

Table 6: Summary of the programs used in the case study, giving the number of lines of code (LOC), nodes within the corresponding GP symbolic tree expression, their input data types and the number of local variables.

same location in the input array A. The fitness is then the sum of errors at individual array positions:

$$f_e(p, t_0) = \sum_{i=1}^{len(A)} |A'[i] - A[i]| \tag{7.5}$$

7.5 CASE STUDIES

To test its effectiveness, eight different C functions are input to the framework. Table 6 summarises their properties: their source code listings are given in Section 7.8.1.

These functions were chosen because they had previously been studied for their execution time and they represent a variety of program structures. They also use simple data types in order to simplify the coevolutionary testing component, and are from freely available and well-documented sources.

Two different implementations of the Triangle Classification program published in McMinn [2004], Miller et al. [2006], and two different implementations of a Bubble-Sort algorithm [Cormen et al., 2001] are used. A recursive implementation of the factorial function [Cormen et al., 2001] is given. The Remainder routine is taken from Sagarna and Lozano [2006]. Finally, from a library of worst-case execution benchmarks [Mälardalen WCET Research Group], the Swi10 and Select (returning the kth order statistic) functions are taken.

The programs include a variety of structures: branching and nesting, loops over arrays, internal state altered multiple times within a function, switch and case statements, use of both temporary variables and arrays, and recursive calls. Two pairs of functions also solve the same problem, which provides for an interesting comparison of their optimisation.

These functions are fairly small and the largest consists of only around 90 lines of code, which is advantageous in performing large-scale experimentation efficiently. A study of the scalability of this approach is important, a matter of future investigation and likely to reflect the scalability of Genetic Programming.

7.6 implementation issues

7.6.1 *Evolutionary Algorithm*

The framework was implemented in Java using ECJ 18 [ECJ, 2009]. The primitives used within the GP algorithm are listed in Table 7. Note that the wrapper types are used to simplify the tree growth mechanisms in Strongly Typed GP (see Section 4.9.1). The table comprises the superset of all primitives required to represent each of the chosen case studies. Using the superset creates a more generally-applicable method. Consequently, we must search a large space of possible trees, and the extensive (necessary) use of Strongly Typed GP results in a representation that does not easily lead to valid programs when carrying out crossover or mutation.

ECJ parameters not detailed here or specified in Section 7.7 were left to the defaults as inherited from `koza.params`, provided with the ECJ distribution. The default method of initialising the population is to use Koza's ramped half-and-half method, with a minimum depth of 2 and a maximum depth of 6 for the half method, minimum and maximum 5 for the grow method.

Three methods are used to populate the next generation: mutation, crossover and reproduction. The probability of reproduction is fixed at 0.1, whereas the balance between the remaining operators is the subject of experimentation. All six methods of mutation ECJ provides are used with equal probability. Each of the three ways of generating the next population selects individuals from the current population using standard tournament selection. When performing crossover or mutation, a terminal or non-terminal is selected with probability 0.1 and 0.9 respectively.

For each problem, an initial set of potential tests containing 2000 individual test cases was generated. The set fulfils the branch coverage criterion for the input function and as such could be generated with any automated software testing technique prior to the execution of the framework, although manually written scripts were used to generate the test cases here. The test case population size is 200, with a further archive of 50 elements. The test case population is partitioned into a number of subsets that depends on the number of branches in the original input program given the expected operational profile.

The instruction count is evaluated on 100 test cases that are randomly sampled at each generation. Integer variables are chosen according to a uniform distribution of values in $\{-127, \ldots, 128\}$. The length of the arrays is uniformly chosen in $\{1, \ldots, 16\}$. Only valid data is used as test cases. For example, where a length is supplied to a particular case study function, the correct length value corresponding to the current input array is given.

With the inclusion of loops and recursion in the primitive set, a limit must be placed on the number of iterations that may occur: a limit of 1000 iterations or recursive calls is used, which is large enough for the case study functions to produce the correct output on all possible inputs.

Category	Name	Number	Description
Arithmetic	$+$, $-$, $*$, / , %	5	Arithmetic operators. % is the modulo operator.
Unary Modification	$++$, $--$	2	Increment and decrement.
Boolean	&&, || , ! , $>$, $\geq$, $==$, $<$, $\leq$	8	Operators to handle Boolean predicates.
Constant	`true`, `false` , 0 , 1 , ..., 9	12	Boolean and integer constants.
Statement	`for`,`while`,`if`,`switch`,`case`, `return`,`skip`,`statement_sequence`, `case_sequence`	9	`skip` is the empty statement. `statement_sequence` and `case_sequence` create sequences of statements.
Variable	`ReadVariable`, `WriteVariable`, `VariableWrapper`, `V_tmp`	≥ 4	Primitives to read and write inputs and local variables.
Array	`ReadArray`, `WriteArray`, `ArrayWrapper`	3	Primitives to read and write array variables.
Other	inline if,`fac`	1	Miscellaneous primitives. *fac* is a recursive call used in the Factorial problem.

Table 7: Primitives grouped by category.

7.6.2 *Non-Compilable GP Individuals*

When evaluating programs in a language such as C pitfalls are encountered that are often neglected in published GP applications, which tend to prefer interpretation rather than direct execution to evaluate a program. For example, consider a protected-division primitive commonly used in the GP literature: implementing this in C would require a macro to check for a division-by-zero, which can be inefficient. One advantage of using a simulator is that when results are successfully achieved, the evolved programs are demonstrably compilable and free of run-time errors for those test cases it has been executed over.

The type constraints on the GP primitives are of a local nature, specifying the return type of each node and the type of parent and children they can have. This is sufficient for most GP applications, but it is not sufficient when modelling a higher level language such as C. In order to evaluate individuals through simulation, we must first translate them to C, and valid GP trees can generate compilation errors once translated. For example, two such problems with the `switch` statement were encountered:

- The indexes of a case within a `switch` statement are not constant, i.e. they cannot be calculated at compile time.
- The indexes of the cases within a `switch` statement are not all unique, i.e. at least one index is used more than once.

Instead of complicating the constraint system to forbid the generation of non-compilable individuals, individuals are punished by setting their fitness to the worst possible value. Compilation and simulation of such an individual are subsequently suppressed.

7.6.3 *Run-time Errors*

Using such a rich subset of the C language in the representation can lead to program errors discovered at run-time. For example, an array may be accessed out of bounds, corrupting memory. This can lead to unpredictable behaviour and the framework must be robust with respect to such eventualities. A similar policy to that of uncompilable programs is followed: the individual is flagged as not having completed every test case and punished via their fitness value.

7.6.4 *Exceeding the Iteration Limit*

Long or infinite iteration is inefficient, costly or impossible to evaluate. The M5 simulator provides support for an upper limit on simulated cycles, and this is used to enforce an execution "timeout" on each individual. The timeout is set to a value twice that of the original input program's instruction count on a representative sample of fitness cases. Individuals exceeding this limit are punished by assigning them the worst possible fitness value.

7.7 EXPERIMENTAL METHOD

7.7.1 *Overview*

Given the proposed framework, there were several issues to investigate: firstly, is it possible to optimise the case studies using the framework? What kinds of optimisation is it capable of producing? These questions are potentially answerable with a single execution of the framework for each case study, with a set of repetitions to ensure robustness. The remaining questions required extensive experimentation. These were:

1. How well can the instruction count of an individual be estimated without executing the simulator?
2. Which of the components and parameters of the framework are important in determining the level of improvement that can be achieved?
3. What is the impact of using a model rather than the simulator to estimate instruction count?
4. How does the use of MOO affect the exploration of the objective space carried out by the algorithm?

7.7.2 *Method*

The experimentation reported consists of the following steps, each one designed to answer the corresponding questions above:

1. Produce a model estimating instruction usage for each problem and evaluate those models using resampling to simulate tournament selection.
2. Carry out a factorial experiment and further comparisons to evaluate the importance of components of the framework.
3. Select parameter settings that performed well in the factorial experiments, and use them to compare the results achieved when employing simulation versus using a model of instruction usage.
4. Collect and analyse data on the impact of using MOO on the exploration of the objective space.

7.7.3 *Factorial Experimentation*

In order to determine which components were important in affecting the performance of the framework, full factorial experimentation was employed [Montgomery, 2006]. This is a robust approach that I have previously demonstrated to be effective when using GP [White and Poulding, 2009], and has previously been applied to investigate the importance of parameters [Feldt and Nordin, 2000]. After initial experimentation the parameters chosen for experimentation and their levels are given in Table 8. As well as the framework components, general GP parameters are included to allow for their impact on the behaviour of other parameter settings.

With 7 parameters at 2 levels and one at 3 levels, there were $2^7 \cdot 3 = 384$ design points per case study. At each design point 30 repetitions

Parameter	Description	Low	High
x_1	Probability of crossover	0.1	0.8
x_2	Population Size	50	1000
x_3	Tournament Size	2	7
x_4	Seeding proportion	0.1	0.9
x_5	Coevolution Enabled?	FALSE	TRUE
x_6	SPEA2 Enabled	FALSE	TRUE
x_7	SPEA2 Archive Size	$0.1x_2$	$0.9x_2$
x_8	Seeding Method	Clone, Mutation or Subtree	
Dependent Parameters			
x_9	Probability of Mutation	0.9 - x_1	
x_{10}	Generations	50 000 / x_2	

Table 8: Experimental parameters.

were executed to allow for variation in the response caused by the random seed. Thus a total of 11520 design points per problem were defined, requiring 92160 experimental runs in total for this experimentation.

7.7.4 *Response Measure*

At the end of an evolutionary run, the best individual is selected from the final population as defined by the weighted sum given in Section 7.4.4. This individual is then tested on new data to evaluate its functional correctness, and is run through the simulator again to measure its non-functional fitness. These are measured using a separate set of 1000 test cases to validate the final output. This does not guarantee that a program is truly semantically equivalent, but it does test the individual against tests independently of the coevolved test set.

The response measure for a single experiment then consists of the semantic and non-functional fitness components of this individual.

7.8 RESULTS

7.8.1 *Example Optimisations*

It is of immediate interest as to *how* optimisations were successfully made by the framework, and here I detail a selection of interesting optimisations taken from the factorial experiments. Note that there are thousands of unique output programs, and those given here are only a small selection. In future work, some form of data mining could be used to extract commonly produced optimisations.

Boxplots of the instruction counts across design points that generated solutions passing the final test set for each problem are given in Figure 29. The variance of the instruction count of optimised programs is quite low for each problem, which demonstrates how robust the technique is to parameter changes. In some cases it is necessary to examine the outliers to see major improvements in instruction count.

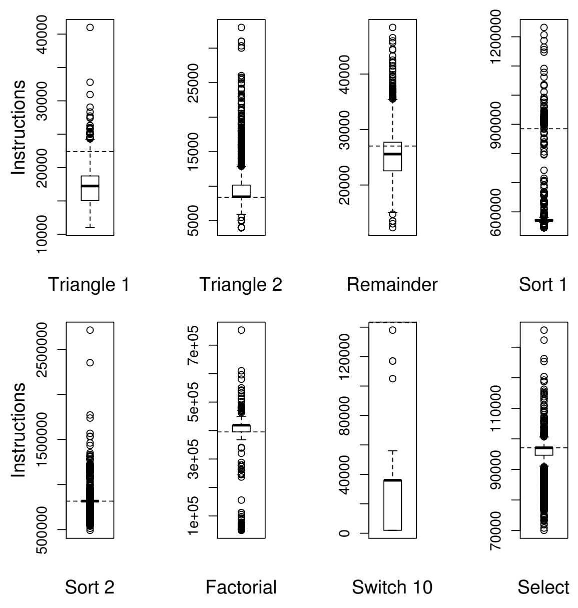


Figure 29: Boxplots illustrating the distribution of instruction counts for valid program outputs. The dashed lines indicates the performance of the original programs.

TRIANGLE1 The triangle classification takes the length of three sides of a triangle as input and returns 1 if the triangle is invalid, 2 if it is scalene, 3 if it is isosceles and 4 if it is equilateral. In the implementation of Triangle1 (see Figure 30), the three inputs are reordered in the three variables (a, b, c). There are three `if` statements controlling three swap operations to achieve this ordering. However, for the algorithm to work, we only require that $a \leq c$ and $b \leq c$, such that a full ordering $a \leq b \leq c$ is not necessary, hence we can discard the first `if` statement. In most of the outputs examined, GP has exploited this by removing the first `if` statement of this swapping process. This does not change the semantics of the program.

GP also found a simple optimisation of swap operations that compilers such as GCC are not able to make. In a swap operation, a temporary variable is employed. After the three `if` statements at the beginning of the code, the temporary variable is not subsequently used. GP optimises the code by using only two assignments for the last swap and then replaces subsequent references to the second variable with references to this temporary variable. In other words, in the last swap operation the command $c = tmp$; is removed, and each successive occurrence of the variable c is replaced by tmp.

The final optimisation found by GP is quite surprising. The following code structure was produced:

```
if(a > c) {
   /* swap a and c */
}

if(b > c) {
   /* swap of b and c, use tmp instead of c */
} else {
   /* classify triangle as original code and return */
```

```
int Triangle1(int a, int b, int c)
{
  int tmp;

  if (a > b)
  {
    tmp = a;
    a = b;
    b = tmp;
  }

  if (a > c)
  {
    tmp = a;
    a = c;
    c = tmp;
  }

  if (b > c)
  {
    tmp = b;
    b = c;
    c = tmp;
  }

  if(a+b <= c)
       return 1;
  else
  {
    if(a == b && b == c)
      return 4;
    else if(a == b || b == c)
           return 3;
         else
           return 2;
   }
}
```

Figure 30: Source code of Triangle1.

```
}

if(a+b <= tmp) {
  return 1;
} else if (a == b) {
  return 3;
} else {
  return 2;
}
```

If the last swap in the original code is executed, then we know that $b > c$. Hence, the triangle cannot be equilateral. GP learns to move the code that checks for the equilateral case into the `else` branch of that swap. Then it replicates the code after that `if` statement to handle the case $b > c$. In this latter case, the triangle cannot be equilateral, and to confirm whether it is isosceles we only need evaluate $a == b$. The execution time is significantly reduced. This is very much a non-trivial optimisation that exploits the information gained in a previous comparison to improve the efficiency of subsequent computation.

TRIANGLE2 GP was able to evolve faster versions for the program Triangle2, given in Figure 31. Analysis of the resulting GP trees did not find any interesting optimisation. This is because GP learnt an *undesired pattern* in the test script used to generate the test cases. This pattern is easy to exploit at the beginning of the code. This optimisation eclipsed other improvements and often led GP to a local (and semantically incorrect) optimum. This underlines the importance of an effective testing system in ensuring semantic equivalence, particularly on boundary conditions.

SORT1 Sort1 implements a naive bubblesort.

The original code is given in Figure 32. The optimised output was:

```
void sort(int[] a, int length) {
    for (; 0 < (length - 1); length--) {
        for (int j = 0; j < (length - 1); j++) {
            if (a[j] > a[1 + j]) {
                k = a[j];
                a[j] = a[j + 1];
                a[1 + j] = k;
            }
        }
    }
}
```

The best solutions found by the framework improved on the original by omitting a single iteration, and using an input variable as a loop counter, rather than using a new variable.

These are sensible optimisations, but perhaps it is a little disappointing that further optimisations were not discovered, such as the use of a *sorted* flag to halt the algorithm once a pass has been completed with a swap. Such an optimisation was given as input to the next problem.

SORT2 Some of the optimisations of this function were similar to those found in Sort1, such as removing the initialisation of a loop counter and using input variables as counters rather than separate individual variables.

```
int Triangle2(int a, int b, int c)
{
  if(a<=0 || b<=0 || c<=0)
    return 1;

  int tmp = 0;

  if(a==b)
    tmp += 1;

  if(a==c)
    tmp += 2;

  if(b==c)
    tmp += 3;

  if(tmp == 0)
  {
    if((a+b<=c) || (b+c <=a) || (a+c<=b))
      tmp = 1;
    else
      tmp = 2;

    return tmp;
  }

  if(tmp > 3)
    tmp = 4;
  else if(tmp==1 && (a+b>c))
        tmp = 3;
      else if(tmp==2 && (a+c>b))
        tmp = 3;
          else if(tmp==3 && (b+c>a))
                tmp = 3;
              else
                tmp = 1;
  return tmp;
}
```

Figure 31: Source code of Triangle2.

```
void Sort1(int[] a, int length)
{
  for(int i = 0;  i < length; i++)
     for(int j=0; j < length - 1; j++)
        if(a[j] > a[j+1])
        {
          int k = a[j];
          a[j] = a[j+1];
          a[j+1] = k;
        }
}
```

Figure 32: Source code of Sort1.

```
void Sort2(int[] a, int length)
{
  int flag = 0;
  int tmp;
  while( flag == 0)
  {
    flag = 1;
    for(int j=0; j< length - 1; j++)
    {
      if(a[j] > a[j+1])
      {
        tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;
        flag = 0;
      }
    }
   }
}
```

Figure 33: Source code of Sort2.

The original input is given in Figure 33. One simple optimisation the system found was to "inline" the first pass of the sort algorithm, giving:

```
for (int j = 0; j < (length - 1); j++) {
  <main body>
}

while (flag == 0) {
  for (j = 0; j < (length - 1); j++) {
    <main body>
  }
}
```

This saves on a single comparison-and-branch, which over 1000 test cases is certainly worthwhile. The system also discovered multiple variants of:

```
void sort(int[] a, int length) {
    int flag = 0;
    while (0 == flag) {
        flag = 1;
        for (j = 0; j < (length - 1); j++) {
            if (a[j] > a[j + 1]) {
                int tmp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = tmp;
                flag = 0;
            }
        length--;
     }
}
```

Note the decrement of *length*. Experienced human programmers may also make this optimisation, another impressive improvement.

```
int Factorial(int a)
{
  if (a <= 0)
     return 1;
  else
     return (a * Factorial(a-1));
}
```

Figure 34: Source code of Factorial.

FAC Fac is a small function (see Figure 34), but one optimisation made by the GP system is to change:

```
if (a <= 0) {
```

to

```
if (a <= 1) {
```

This saves one recursive call for test cases containing positive inputs.

A more interesting optimisation is the exploitation of overflow behaviour in Java. Individuals were evaluated in Java for the most part, as the large-scale experimentation involved favoured the more efficient interpretation rather than simulation of individuals. Thus the system ensured semantic equivalence with the Java version, whilst targeting a model of execution time on an embedded processor. This can lead to unforeseen issues if the Java interpreter does not match the semantics of compiled C. Nearly all such cases were eliminated, and continuous validation of results highlighted the exception of one such behaviour when optimising the factorial function.

Inspecting some of the most efficient individuals, it quickly became apparent that evolution had "decided" it was beneficial to add conditional statements such as the following three examples:

```
if (!((2 * (9 + 8)) <= V_a))

while (V_a < (7 * (1 + 4))) {

if (V_a <= (8 + (8 + (8 + 9)))) {
```

Following this statement would be the usual factorial function. Each constant being compared to the input is 33, 34 or 35. Clearly, not calculating fac(33) or fac(34) was an optimisation in terms of timing – but surely this would break the semantics of the program?

However, fac(34)=0 according to the original program. An overflow within Java resulted in a zero return value. Therefore, for any value $n \geq 34$, fac(n)=0 because a multiplication by fac(34)=0 will occur during the computation of fac(n). The GP system correctly discovered that cycles were being wasted in these cases, and by adding this `if` statement, it enabled the code to fall through to the default "return 0" at the end of the test harness.

REMAINDER The source of the Remainder function is given in Figure 35. Most of the optimisations of this (somewhat inefficient) code simply use the % modulo operator, as expected. One interesting optimisation is the removal of the first `if` statement, which checks for a zero value of a, and directly returns -1 for that value. Removing this check does not

```
int Remainder(int a, int b)
{
  int r  = -1;
  int cy = 0;
  int ny = 0;

  if (a==0);
  else if (b==0);
       else if(a>0)
              if(b>0)
                 while((a-ny)>=b)
                 {
                   ny=ny+b;
                   r=a-ny;
                   cy=cy+1;
                 }
              else // b<0
                 while((a+ny)>= ((b>=0) ? b : -b))
                 {
                   ny=ny+b;
                   r=a+ny;
                   cy=cy-1;
                 }
            else  // a<0
              if(b>0)
                while( ((a+ny)>=0 ? (a+ny) : -(a+ny))
                    >= b)
                {
                  ny=ny+b;
                  r=a+ny;
                  cy=cy-1;
                }
              else
                while(b>=(a-ny))
                {
                  ny=ny+b;
                  r= ((a-ny)>=0 ? (a-ny) : -(a-ny));
                  cy=cy+1;
                }
  return r;
}
```

Figure 35: Source code of Remainder.

```
int Swi10(int a)
{
  for (int i=0; i<10; i++)
  {
    switch (i)
    {
      case 0: a++; break;
      case 1: a++; break;
      case 2: a++; break;
      case 3: a++; break;
      case 4: a++; break;
      case 5: a++; break;
      case 6: a++; break;
      case 7: a++; break;
      case 8: a++; break;
      case 9: a++; break;
      default: a--; break;
    };
   }

   return a;
}
```

Figure 36: Source code of Swi10.

change the semantics (i.e., the return value will still be -1, returning later in the function). This is an interesting optimisation, because it exploits the expected input distribution, the operational profile of the code. The original code is only useful when the input a is zero, which is not a common occurrence as its input domain is [-127,128]. Had the input domain been (for example) [-1,2], then the optimisation would not have been worthwhile. It is probably the case that this simple method of optimisation might be applied to many other programs.

SWITCH 10 Switch 10 (see Figure 36) was chosen deliberately to see if our system could optimise it in the obvious way. Indeed, I found that the system was able to find the following:

```
  return 10 + a;
```

A minimal solution, containing just this code, was found many times by the system. This is a non-trivial optimisation that GCC -O2 was unable to achieve, a satisfying result.

SELECT Select, given in Figure 37 is the longest and arguably the most complicated function. Many of the output optimised programs were large, and this made it time-consuming to analyse them. Perhaps the introduction of a sophisticated parsimony method such as that proposed by Poli and McPhee [2008] may improve the readability of the outputs.

Nevertheless, there are some common trends to the optimisations. The first optimisation is a check for negative values:

```
if (a >= k) {
  flag = 1;
}
```

```
int Select(int[] arr, int k, int n)
{
  int i=0,j=0,mid=0,a=0,temp=0;
  int flag = 0, flag2 = 0;
  int l=1;
  int ir=n;

  while (flag == 0)
  {
    if (ir <= l+1)
    {
      if (ir == l+1)
        if (arr[ir] < arr[l])
        {
          temp=(arr[l]);
          (arr[l])=(arr[ir]);
          (arr[ir])=temp;
        }

      flag = 1;
     }
     else if ( flag == 0)
          {
            mid=(l+ir) / 2;
            temp=(arr[mid]);
            (arr[mid])=(arr[l+1]);
            (arr[l+1])=temp;

            if (arr[l+1] > arr[ir])
            {
              temp=(arr[l+1]);
              (arr[l+1])=(arr[ir]);
              (arr[ir])=temp;
            }

            if (arr[l] > arr[ir])
            {
              temp=(arr[l]);
              (arr[l])=(arr[ir]);
              (arr[ir])=temp;
            }

            if (arr[l+1]> arr[l])
            {
              temp=(arr[l+1]);
              (arr[l+1])=(arr[l]);
              (arr[l])=temp;
            }

            i=l+1;
            j=ir;
            a=arr[l];
```

Figure 37: Source code of Select (continued overleaf).

```
            while ( flag2 == 0)
            {
              i++;
              while (arr[i] < a)
                i++;
              j--;
              while (arr[j] > a)
                j--;

              if (j < i)
                flag2 = 1;

              if (flag2 == 0)
              {
                temp=(arr[i]);
                (arr[i])=(arr[j]);
                (arr[j])=temp;
              }

              arr[l]=arr[j]; arr[j]=a;

              if (j >= k)
                ir=j-1;
              if (j <= k)
                l=i;
             }
         }
  return arr[k];
}
```

Figure 37: continued.

By placing this at the start of the program, no iterations are made and the function quickly returns.

The second optimisation is that the conditional statement in the `else` part of the main `if` statement can be removed:

```
else if ( flag == 0) {
```

This is replaced with:

```
else {
```

This can be done because `flag` is tested in the `while` statement previously.

The original solution is also slightly inefficient in one of its exchanges, the framework found this issue and removed the inefficiency. Observe:

```
temp=(arr[mid]);(arr[mid])=(arr[l+1]);(arr[l+1])=temp;

if (arr[l+1] > arr[ir]) {
temp=(arr[l+1]);(arr[l+1])=(arr[ir]);(arr[ir])=temp;
}
```

The first assignment of the second exchange is unnecessary, as `temp` already contains the value in `arr[l+1]`. This was exploited by many output programs.

7.8.2 *Overview of Improvements*

It is interesting to consider the range of improvements the framework made. Table 9 gives an overview of the results that were achieved during the factorial experimentation described in Section 7.8.4.

The original programs were compiled using GCC's -O2 optimisation level (as it was the most sophisticated level I am able to profile using the simulator), and run through the 1000 validation tests used by the factorial runs to test output individuals. The number of instructions consumed by each of the original programs is given in Table 9. This is the baseline by which improvements must be measured.

The table shows the number of best individuals successfully completing with zero errors in their final fitness evaluation. It then lists those classified as "valid" after checking them on a validation set (this does not necessarily guarantee semantic correctness), and those passing the tests that also constituted an improvement on the original program's performance.

Note that these figures simply summarise the experimentation, over a range of parameter values and I would not expect good performance at each of the 384 design points (parameter settings). As an indicator of the diversity of solutions produced for each problem, the number of unique instruction counts are given.

7.8.3 *Modelling Instruction Count*

In order to construct a model, it was necessary to gather data from an evolutionary run, and so one evolutionary run for each case study was performed, with simulation enabled. The number of instructions an individual used within the simulator and the count of the high level primitives evaluated when running through the ECJ interpreter were logged. Parameter settings were based on ECJ defaults for the run, and manually selected settings were used where defaults were

	Output Programs				Instruction Counts			
Problem	**Successful**	**Valid**	**Improvement**	**Unique**	**Original**	**Minimum**	% Improvement	**Median**
Triangle1	11507	2203	2112	1444	22402	10996	50.9	17214.5
Triangle2	11433	1981	524	187	8380	4000	52.3	8000
Remainder	11070	4805	3184	526	27025	12318	54.4	23318
Sort1	11502	3131	2708	128	884842	545878	38.3	569517
Sort2	11496	5271	1831	348	815882	492217	39.7	790062
Factorial	11514	3374	142	51	395189	49664	87.4	278700
Switch 10	11520	4763	4763	34	143000	2000	98.6	36000
Select	11506	3653	1206	507	97077	70085	27.8	90201.5

Table 9: Summary of factorial experiments.

Problem	Total	Unique	No Run-time Errors	Completed Successfully
Triangle1	62500	13847	13835	13833
Triangle2	62500	12960	12959	12959
Remainder	62500	14065	14061	14030
Sort1	62500	8220	8220	8220
Sort2	62500	7972	7972	7972
Factorial	62500	9465	9463	9455
Switch	62500	3047	3037	3034
Select	62500	20521	20520	20519

Table 10: Individuals sampled when constructing models.

not available or appropriate – the parameter settings themselves being of less importance in model-building than when actually optimising software.

These experiments used a population size and generation limit both set to 250, effectively sampling 62500 points in the search space. Increasing the sample size any further would have taken individual evolutionary runs beyond the scope of weeks of compute time, which quickly became impractical. An evolutionary run was preferred as opposed to systematically sampling the space in order to ensure the sample was representative of the individuals likely to be encountered during a run.

The sample is restricted in its generality by two factors: firstly duplicate data where the same program is sampled more than once (a disadvantage of using data from an actual run) and secondly programs where full evaluation was not possible. The former is to be expected, particularly with seeding methods that heavily favoured the introduction of programs identical or similar to the original input function. The latter was caused by problems such as run-time errors in individual programs and iteration limit timeouts. A summary of data at the larger sample size illustrate the extent of this issue, as given by Table 10.

Applying standard least squares linear regression to construct the model, it was observed that negative coefficients resulted. It is intuitive that this may cause problems: the evolutionary framework might exploit this by adding extra instructions with negative coefficients and this would improve the fitness of an individual provided that it did not interfere with its functional behaviour. In other words, there was a danger of actually encouraging the emergence of bloat! Exploratory runs confirmed that negative coefficients quickly lead to a great deal of program bloat. Therefore, the regression was repeated with coefficients constrained to positive values only, using the R `nnls` library [R, 2009].

Some of the coefficients were zero, usually because the instructions concerned had not been sampled sufficiently to accurately estimate their cost. This is a limitation of the modelling approach, given a limited availability of compute power, and I accept these inaccuracies here as part of the expense of approximation.

It is now possible to evaluate the accuracy of modelling. By sampling a selection of individuals of tournament size n, then carrying out tournament selection on that group using firstly the instruction count

Problem	Tournament Size 2	Size 5	Size 7
Triangle1	66.1%	43.4%	41.4%
Triangle2	55.3%	29.0%	23.9%
Remainder	75.6%	64.6%	62.1%
Sort1	75.7%	73.8%	74.5%
Sort2	79.0%	73.4%	74.4%
Factorial	80.8%	69.3%	63.0%
Switch	74.1%	61.8%	57.1%
Select	72.0%	53.4%	49.8%

Table 11: Accuracy of modelling.

from the simulator, and secondly the estimate from the model, the accuracy of a model can be measured by comparing how many times the two choose the same individual. The results are in Table 11, based on a sampling of 10,000 simulated tournament selections in each case. This modelling in effect adds noise to the fitness function.

The model proved to be sufficient to achieve the optimisations previously described, and a full comparison to simulation is given in Section 7.8.5. The algorithm was sufficiently robust to accept this level of noise in the selection mechanism.

7.8.4 *Important Components and Parameters*

The results of the factorial experimentation can be used to analyse the behaviour of the framework.

The priority of a practitioner must be to produce error-free software first, and solutions that are time-efficient as a secondary objective. Therefore, for each problem the ten treatments with the greatest probability of producing error-free solutions were selected, as measured by the proportion of zero error outputs. A new set of 30 runs for each of these ten treatments for each problem was then recorded by repeatedly running the framework with each treatment and recording the outcome of successful runs (nearly all runs were successful). The performance of these settings in our repeated results had similarly low error rates and produced completely valid programs in all but six cases.

A boxplot of the 30 repetitions for the ten best treatments chosen for Triangle1 is given in Figure 38. There is little variety in the ability of the treatments to find good optimisations. These boxplots are representative of other problems: the top ten treatments generally give similar results (with a few treatments having much higher variance). This is promising, as it adds weight to the argument that the framework is robust to its parameter settings.

The best parameter settings are chosen based on their medians, for use in Section 7.8.5. These settings cannot be regarded as optimal settings in a general sense, only that they performed well in the full factorial on the case studies. These settings are listed in Table 12. Taking these top ten treatments, we may examine them to assess the impact of specific parameter settings.

Parameters	Triangle1	Triangle2	Remainder	Sort1	Sort2	Factorial	Switch 10	Select
Probability of crossover	0.8	0.1	0.8	0.1	0.1	0.1	0.1	0.8
Population Size	1000	1000	50	50	50	50	50	1000
Tournament Size	2	2	7	7	7	7	7	7
Seeding proportion	0.9	0.9	0.1	0.1	0.1	0.1	0.1	0.9
Coevolution Enabled?	Yes	Yes	Yes	No	No	No	No	Yes
SPEA2 Enabled?	No	No	No	No	No	No	No	No
SPEA2 Archive Size	100	900	5	5	5	5	5	100
Seeding Method	Mutation	Clone	Clone	Clone	Clone	Clone	Clone	Clone

Table 12: Best parameter settings for each problem.

Table 13 lists the number of times the components were set to the high value in the best treatments for each problem. Note that the seeding method had three levels, hence in that case the frequency of the low/medium/high values are given. The main patterns we can observe from this table are:

- Most parameters are heavily problem-dependent, that is sometimes the features of the framework are helpful and sometimes they are not. The problem-specific nature of the parameters is underlined by the similarity between the settings for the two problems optimising sort functions.
- Apart from Select (the largest of the problems), the best settings did not use SPEA2. I conjecture that this is because SPEA2 encourages exploration that is unnecessary to achieve good improvements in execution time for functions of this size. This behaviour is investigated in more detail in Section 7.8.6.
- The third type of seeding method, subtree seeding, is rarely used. Thus it does not appear that "building block reassembly" is one of the mechanisms our framework uses to find optimisations.
- Generally a high value of seeding proportion is preferred. This confirms the expectation that seeding would be important: that GP is not creating solutions from scratch.
- Coevolution is favoured, particularly in those problems that have boundary conditions. It is likely that coevolution is able to find key discriminating test cases more efficiently than random testing.

It may be possible to separate problems into those most suitable for optimisation using mutation, and those more susceptible to crossover. The former involve smaller, localised optimisations that may be applied stepwise, i.e. their beneficial effect is additive. The latter require more radical restructuring. This may explain the modal pattern of best settings seen between Triangle, Remainder, Select as one group and the Sort, Factorial and Switch problems as another.

7.8.5 *Comparing Modelling to Simulation*

To analyse the impact of using full simulation rather than modelling of a program's instruction usage, I used the parameter settings given in Table 12 and ran 20 repetitions of each, firstly retaining the model as the method of evaluation, and secondly using the full simulator. The latter runs took days or weeks, as full simulation is computationally expensive.

Boxplots comparing the resulting distributions of instruction count are shown in Figure 39. I then performed a nonparameteric Mann-Whitney rank-sum test for significance between the distributions for each problem. In cases where a significant difference was found at the 0.05% level, I carried out a further test of effect size to determine the *scientific significance* or importance of the difference, and all figures for these tests are given in Table 14.

Surprisingly, there are only four problems where there exists a statistically significant difference at the 0.05% level. Taking significant effect size for the Vargha-Delaney A statistic [Vargha and Delaney, 2000] at

Parameters	Triangle1	Triangle2	Remainder	Sort1	Sort2	Factorial	Switch 10	Select
Probability of crossover	10	7	10	0	0	0	0	4
Population Size	4	10	4	0	0	0	0	4
Tournament Size	7	4	5	10	10	10	10	4
Seeding proportion	6	8	7	6	6	6	6	8
Coevolution Enabled?	10	10	8	4	3	4	4	4
SPEA2 Enabled?	0	0	2	0	0	0	0	6
SPEA2 Archive Size	4	5	5	5	5	5	5	5
Seeding Method	4/6/0	5/5/0	5/5/0	7/3/0	6/2/1	7/3/0	7/3/0	5/2/3

Table 13: Number of top ten treatments containing high values.

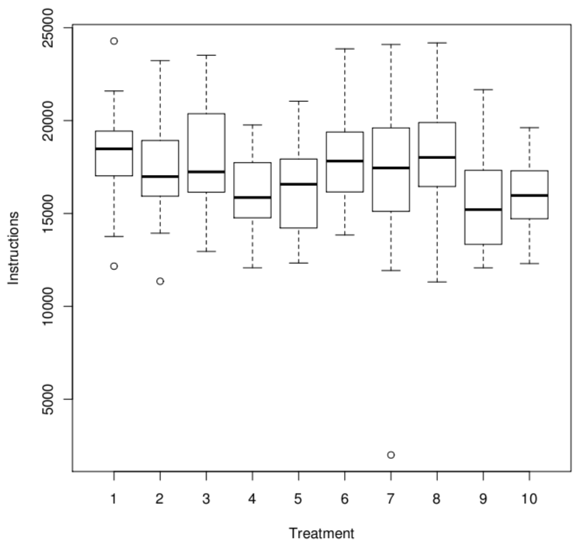


Figure 38: Instruction counts for the ten best treatments of Triangle1.

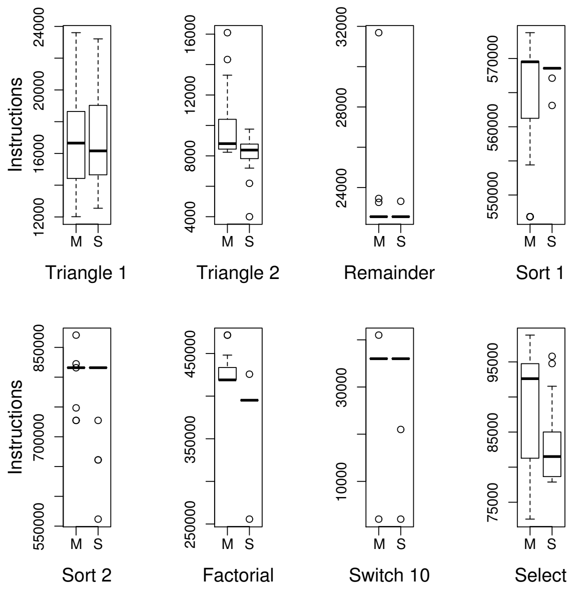


Figure 39: Comparing instruction counts using modelling versus simulation.

Problem	Rank-Sum P-Value	Vargha-Delaney A Statistic
Triangle1	0.892	n/a
Triangle2	0.006	0.245
Remainder	0.299	n/a
Sort1	0.0203	0.295
Sort2	0.423	n/a
Factorial	6.89e-08	0.0375
Switch 10	0.350	n/a
Select	0.0206	0.285

Table 14: Comparing instruction count distributions using modelling versus simulation.

less than 0.36 or greater than 0.64, I can state that these four differences are also *scientifically significant*, i.e. of a magnitude that we should be interested in. As expected, those significant differences all represented cases where the use of a simulator improved performance.

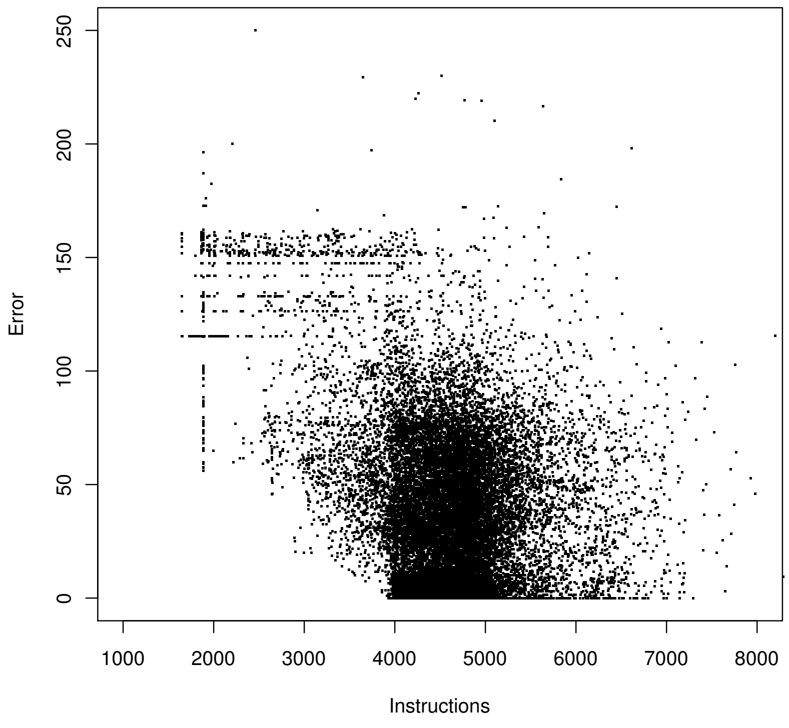
Whether or not to employ simulation therefore depends on the goal of the practitioner. For further research along the lines of this thesis, modelling is an efficient alternative to simulation that still provides the opportunity to achieve large optimisations. Similarly, incorporating large-scale search into software design in the medium term would appear to favour the option of modelling over simulation due to the computational requirements of the simulator. Certainly, manually designed or more sophisticated automated modelling methods may improve on the simple linear model used in this work.

When optimising more complex programs, or taking into account detailed machine-level behaviour such as cache access and misses, it is recommended that a simulator is employed. Whilst a simple approach to modelling can achieve impressive results, it is unlikely to be able to do so as the relationship between high-level source code and runtime behaviour becomes more complex. Modelling is not an option, for instance, in Chapter 8.

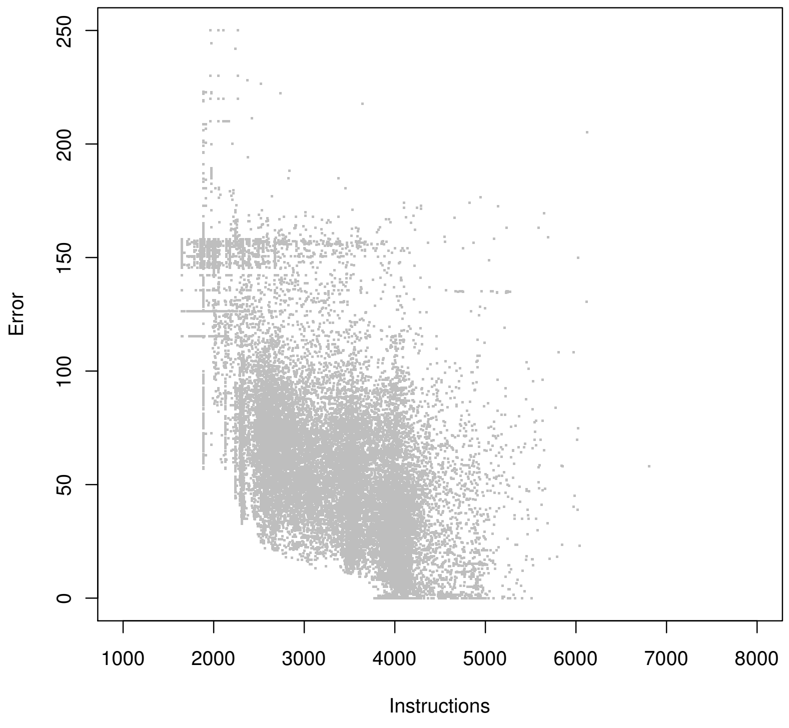
7.8.6 *Exploration using MOO*

It was previously speculated that taking a Pareto-based approach using the SPEA2 algorithm would increase the exploration of the search space, such that small highly fit subtrees may be recombined efficiently to create improved solutions. To assess the impact of using SPEA2, the best parameter settings from the factorial experiments (all of which did not use MOO) were compared to the same settings with MOO enabled, over 30 repetitions. During the runs, the pair of values (error, instruction count) for each individual evaluated were logged.

Figure 40 compares the exploration of the *objective* space between the two pairs of settings, for a single repetition of Triangle1. Other repetitions produced very similar results. Surprisingly, the difference is not great. The SPEA2 method explores more points closer to the y axis, but the difference is limited and the error values on those extra points explored are quite large. Thus it appears that SPEA2 is not exploring small, highly-fit programs.



(a) Weighted Fitness



(b) Pareto-based Fitness

Figure 40: Exploration of the objective space for Triangle1.

Problem	Sampled	Unique Weighted	Unique SPEA2
Triangle1	1500000	692170	720198
Triangle2	1500000	755688	580429
Remainder	1500000	166782	912945
Sort1	1500000	241206	521717
Sort2	1500000	194215	626644
Factorial	1500000	155401	637414
Switch 10	1500000	37150	256402
Select	1500000	435618	682031

Table 15: Unique objective values sampled by weighted and SPEA2 methods.

To summarise the data, Table 15 lists the number of unique points in the objective space sampled across the thirty repetitions for each problem, both for the weighted approach (SPEA2 disabled) and with SPEA2 enabled. SPEA2 samples far more unique points in general, although Triangle2 is an exception.

I conclude that SPEA2 or other Pareto-based methods are most likely to be useful in optimising larger programs, or at least that diversity-maintaining functions should be used when trying to achieve a scalable optimisation method. For smaller programs, of the size examined here, it is unlikely to be of use.

7.9 LIMITATIONS

The framework has the following limitations:

- Software testing cannot prove that a program is free of errors [Myers, 1979]. The transformations applied to the programs are not semantics-preserving, and thus cannot guarantee that the output of the framework is semantically equivalent to the input program. Therefore, output must be manually verified in problems of Boolean acceptability.
- The space of all test cases used to validate the program's semantics is constructed before the search. Potential test cases are chosen based on structural criteria (e.g., branch coverage) of the input program. However, evolved programs can have different control flow and different boundary conditions, and thus a test set designed for the input program may not be appropriate later in the evolutionary search. An alternative would be to generate new test cases at each generation. This could be achieved by creating new test cases based on structural coverage of the best individual in the population.

7.10 FUTURE WORK

The framework presented is still prototypical, and there are many directions future research could take:

- Scalability is an important factor that needs to be studied in more detail. Will this approach be effective only on relatively small functions, or can it scale up to larger systems? Even if its scalability is limited, it may still be useful because it can obtain types of optimisations that current techniques cannot.

- Seeding strategies are important to help the search process to focus on promising regions of the search space by exploiting useful information from the input program. However, seeding strategies can have a drastic impact on the diversity of the program population. Search operators that have been designed for randomly initialised populations (e.g., Koza's single point crossover) may not be the most effective for greedy seeding strategies. Tailoring the search to the problem by using alternative mutation operators might be more effective.

- Analysing the outputs of the framework was a daunting task: *there were literally thousands of different optimised programs produced*, and each of these outputs contain potentially useful information. By applying data-mining methods, common optimisations could be isolated and examples presented to an engineer. Furthermore, mining results from a variety of problems could lead to generic templates of optimisation methods that could be easily incorporated into a conventional compiler.

7.11 CONCLUSION

I have presented a novel framework to improve non-functional criteria of software. In particular, I concentrated on instruction count, although other criteria could be considered.

Transformations of the programs that do not guarantee the preservation of the original semantics were used. This enables the creation of new versions of the programs that could not be obtained with semantics-preserving operators. To address the problem of maintaining semantic equivalence, the evolving programs were subjected to intensive testing. As the original input program can be used as an oracle, it is possible to generate an arbitrary number of test cases limited only by computational resources and time constraints.

For a set of case studies, interesting useful and non-trivial optimisations were found, in a robust manner with respect to the parameter settings of the algorithm. This original approach to automated generation of suggested optimisations has great potential. It could be used to aid developers in fine-tuning optimisation, as well as providing a method of discovering generalisable optimisations that could be incorporated into a compiler.

8 FINE-GRAINED CONTROL OVER TIMING

8.1 INTRODUCTION

In the preceding chapters I have focused on resource efficiency by minimising the resources that a program consumes. I have attempted to locate efficient trade-offs such that the minimal amount of functionality is sacrificed in decreasing resource consumption where a trade-off must be made.

In this chapter, I investigate just how finely a non-functional property (timing) of code can be controlled using Genetic Programming. There are several reasons for investigating this. Firstly, if we are to make use of the smallest amounts of energy, then we must be able to budget carefully. Secondly, the ability to fine-tune exactly how much of a resource is consumed by software enables GP to manipulate the resource consumption as a *side-channel*. The latter has immediate application in the realm of embedded systems security, where side-channel analysis is a genuine threat to the integrity of a system.

There are other potential applications for fine-grained control, some more ambitious than others. This chapter explores the manipulation of code's run-time in a general sense, illustrating how it could be applied alongside proof-of-concept examples. It is worth emphasising that what I present here concerns the timing properties of software, but could be applied to any other measurable property such as power consumption or memory usage.

8.2 RESOURCE-CONSUMPTION AS A FUNCTION

Can we control the execution time of a program p on input x, $T(p, x)$, such that T can be an arbitrary function of x? Such control might allow us to produce more efficient solutions to some problems than possible with functional computation alone, and also create software with useful security properties.

In the following sections, I demonstrate the difficulty of trying to manually control the low-level timing properties of software, by attempting to create code that has a specific time complexity relationship. This demonstrates the potential superiority of GP against manual design alone as a tool for such tasks.

8.3 EVALUATING LOW-LEVEL TIMING

In order to evaluate individuals, I evolve them in the ECJ 19 Toolkit [ECJ, 2009] before writing them out as C code, and compile them together with a test harness using a GCC cross-compiler. The resulting executable is then run within the M5 Simulator [Binkert et al., 2006].

The M5 simulator is targeted for an Alpha architecture, using the default parameters of the simulator. There is a crucial difference in this work, however, because the *absolute* values of resource consumption are of interest, rather than relative differences between individuals.

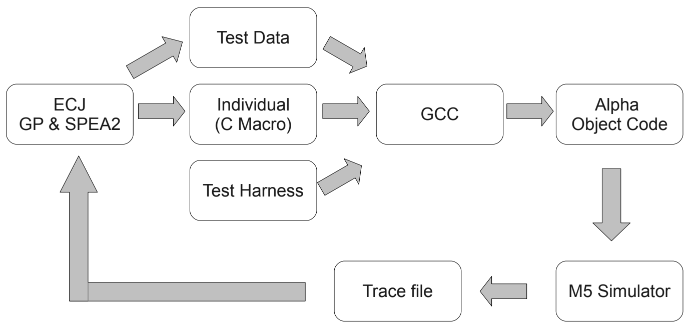


Figure 41: Evaluation of fine-grained timing.

The absolute values are required because I wish to use GP to control the fine-grained properties of software's interactions with hardware. As such, it is necessary to simulate all details of the system, including cache operation and instruction pipelining using full out-of-order simulation. This is consequently an expensive fitness function (up to 20 minutes per individual in this chapter!) with the given experimental arrangement, and limits the size of the examples I may investigate.

The processor is again an Alpha-ISA model of a DEC Tsunami system, at a default speed of 2GHz and 512MB main memory with two sets of cache: 32KB data and 64KB instruction level 1 and 2MB level 2. The speed of the processor, and size of the main memory are not important in our evaluation and are left at their defaults for convenience. Changing cache values would yield different results to those reported here, which is part of the reason for investigating GP in controlling the interactions of such subsystems.

An individual's cycle consumption is measured by surrounding it with NOPs and parsing M5's output tracefile to extracted the simulated ticks. A test harness for each problem is handwritten, and the current test data for that individual at a specific generation is output from ECJ as hardcoded data in a C source file to be linked to the harness. A diagram of evaluation is given in Figure 41.

An example test harness is given in Figure 42, and an example individual in Figure 43. Each individual code segment to be tested is written out as a macro using the C `#define` statement. This allows the evaluation of small code fragments in the context provided by the test harness and eliminates the effects of direct function calls.

Where handwritten solutions are examined in the following sections, the C macro has been manually written, and the rest of the evaluation method remains unchanged to ensure a fair comparison with evolved solutions.

8.4 DESIGNING SPECIFIC COMPLEXITY

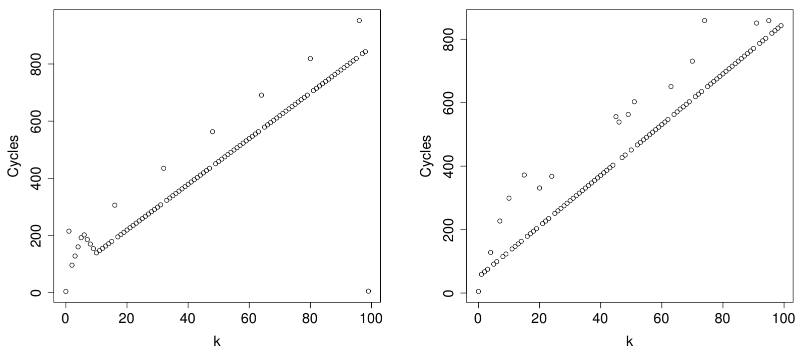
It is second nature to think about a program in terms of its resource consumption complexity: quadratic is good, linear is better, exponential is undesirable. For a moment, let us focus only on the complexity of a program rather than its purpose. Consider a program that has linear complexity with respect to quantity n. Quantity n may be a representation of problem size, in which case we assume the conventional notion of the complexity of the program, i.e. the commonly assumed interpre-

```
#include <stdio.h>
#include <stdlib.h>
#include "testdata.c"

int main(int argc, char** argv) {
        cut();
        // sim. modified to only trace this function
        return 0;
}

cut() {
        int i;
        float tmp, c;
        printf("Test Results\n");
        fflush(stdout);
        for (i=0;i<sampleSize;i++) {
                tmp = 0;
                asm("nop");
                cut_inline(input[i]);
                asm("nop");
        }
        printf("Tests Successful\n");
        fflush(stdout);
}
```

Figure 42: Example test harness.

```
#define cut_inline(k) \
        if (tmp) { \
                for (c=0;c<k*tmp;c++) {   \
                        ; \
                }; \
        } else { \
                ; \
                tmp = k - 1; \
        }

#include "simple_test_randomised.c"
// test harness
```

Figure 43: Example individual written out to C.



(a) Ordered test data (b) Randomised test data

Figure 44: Test results for handwritten linear solution.

tation of big "O" notation etc. If we replace the concept of problem size with "input value", we now have a program that has a linear relationship between the *value of its input* and its resource consumption. This is related to the theory of pseudo-polynomial time algorithms.

If I wish to construct code that may have this behaviour, such that $T(p,k) = mk + c$, I could suggest such a program without regard to its functionality as thus:

```
float tmp = 0;
while (tmp < k) {
        tmp++;
}
```

The functional behaviour of this code is irrelevant at this stage: I only wish to repeatedly carry out some quantity of work in a linear relationship to the numeric input value. I have used `float` rather that `int` here to maintain generality throughout the following sections, and because floating point operations have interesting timing properties.

As an example, I now test this program on the inputs $k = 0, 1, \dots 99$. Figure 44a gives a graph of the results. Note that we do not experience a perfect relationship, due to the interaction between the test program and machine state. The test cases are executed within a loop, and the previous test case will affect the instruction count of the current test case due to its effect on machine state. The primary cause of the outliers is data cache misses for the input values, such that the comparison operation causes a delay prior to the next instruction. As such, the context of the machine state is determined by the test harness: more on this Section 8.5.1. If we randomise the order of the inputs, it is possible to measure the relationship in a more robust manner. The result is given in Figure 44b. Here we have even larger outliers, but they are spread throughout the input range rather than concentrated at the beginning.

We can quantify the relationship by estimating Pearson's correlation coefficient for this sample:

$$r(k,T) = \frac{\sum_{i=1}^{n}(k_i - \bar{k})(T_i - \bar{T})}{\sqrt{\sum_{i=1}^{n}(k_i - \bar{k})^2}\sqrt{\sum_{i=1}^{n}(T_i - \bar{T})^2}} \tag{8.1}$$

where k_i is the ith test case and the time T_i is the time taken to execute a code fragment p on this input. A perfect positive correlation would be 1,

Objective	Find a program $p^*(k)$ such that $\Theta(f) = mk + c$
Terminal set	k, tmp
Function set	FixedLoop, $\leq$, increment tmp, $+$, $*$, $-$, $/$, if, sequence, skip, update tmp
Fitness cases	$k = 0, 1, \ldots, 99$
Fitness function	$r(k, T)$
Parameters	Initial tree depth = 3, generations = 10, population size = 20, prob(xo) = 0.9, prob(mutation) = 0.1
Initialisation	Koza's ramped half-and-half method

Table 16: Experiment A: Settings to evolve linear time behaviour.

Function	Returns	Arguments
k, tmp	float	-
FixedLoop	statement	float, statement
$+$, $*$, $-$, $/$, $\leq$	float, float	-
if	statement	float, statement, statement
Increment tmp, skip	statement	-
Update tmp	statement	float
seq	statement	statement, statement

Table 17: Function return and argument types.

a negative correlation -1 and 0 no correlation. The correlation coefficient for this manual attempt at linearity is 0.978, and by no means a perfect correlation due to the subtle interaction with the program's context. An interesting question, therefore, is: *can evolution find a better solution?*

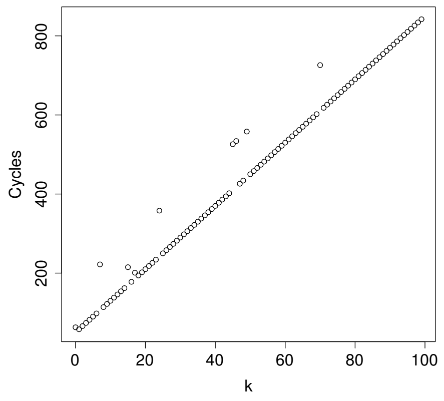
8.4.1 *Experiment A: Linear Behaviour*

We must decide on a function set, and a description of the experiment is given in Table 16. Note that the maximum number of cycles is limited to a similar value to that used by the handwritten solution, in order to prevent GP from finding much larger solutions to simply mask the "noise" of the program context. Also, the randomised order of tests is varied at each generation, to prevent overfitting. Note that `FixedLoop(n)` will loop for n iterations, using a C `for` loop. The function `Update tmp` will set `tmp` to the value of its argument.

I employ Strongly Typed Genetic Programming with this function set, and the types of each function and its arguments are given in Table 17. Note that I did not introduce a specialised Boolean type. This reduces the constraints on the search process by simplifying the type system. Also, the simple FixedLoop avoids both introducing a high-arity function (see Section 4.3.3 as to the impact of high arity functions on the search space size) and the problem of infinite loops in evaluation, which leads to expensive timeouts.

```
if (k) {
   for (c=0;c<k;c++)
        {
   };
} else {
   tmp++;
   tmp++;
   tmp++;
}
```

(a) Source code



(b) Time against input

Figure 45: Evolved linear solution.

Genetic Programming is indeed able to locate a better solution in 10 generations with a population of 20, and this individual has a correlation coefficient of 0.993. A graph of this relationship and the code of the individual evolved is given in Figure 45.

The key differences between the evolved and handwritten solutions are:

- When $k = 0$, work will still be performed in the evolved version, namely testing for this case, and moving through further branches.
- When $k > 0$, an additional `if` statement evaluation and branch will be performed.

If we examine the trace output for this evolved solution, it includes exactly the same instructions executed in the manual solution, and adds some to either side of the loop. This effectively "smooths" the response (time taken) by suitably padding the instruction pipeline during cache access, as well as increasing the time taken for $k = 0$, i.e. setting a larger value for the intercept.

Note that we can now (approximately) calculate $f(k)$ by running the program and observing the time taken to execute, where in this case $f(k) = mk + c$.

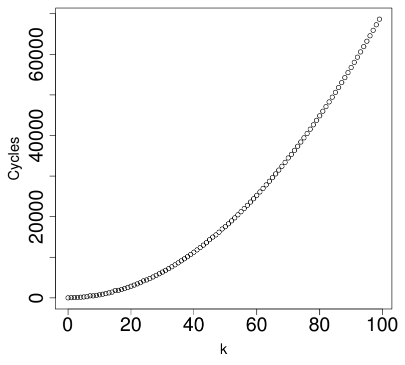
8.4.2 *Experiment B: Quadratic Behaviour*

What if we wish instead to create a nonlinear relationship such as a quadratic curve with no linear term? We may try a manually written solution as given in Figure 46.

This solution takes the general approach of calculating $f(k)$ prior to repeating the strategy of looping to perform a multiple of a minimal unit of work (increasing c). This relationship appears to be a great improvement on the manual attempt at a linear solution, but we must

```
tmp = k * k;
for (c=0;c<tmp;c++);
```

(a) Source code



(b) Time against input

Figure 46: Test results for handwritten quadratic solution.

quantify the relationship to be sure. We can estimate the derivative of the evolved function as:

$$f(k+1) - f(k) = m(2k+1) \tag{8.2}$$

Thus we can measure the correlation between the differences of successive timings and k in order to evaluate the fit of the quadratic relationship. For the handwritten solution above, the correlation is 0.962. Can we improve on this using evolution? I first used the correlation measure as a fitness function, but after some manual experimentation and analysis it became apparent that rewarding a certain amount of first-order correlation is also desirable in order to guide the search, and hence a weighted fitness function was used:

$$F(p) = w_0 \cdot r(k, T(p,k)) + w_1 \cdot r(k, T(p,k+1) - T(p,k)) \tag{8.3}$$

A simple selection of 0.2 and 0.8 for w_0 and w_1 respectively was sufficient to guide the search to a successful solution, found using the same parameter settings as given in in Table 16. The best individual evolved had a correlation of 0.967 between the input and the estimate of the derivative and is given below:

```
for (c=0;c<(k * k );c++) {
        tmp = tmp;
};
```

This is a very modest improvement over the manually written version.

8.5 time as a functional output

In the preceding section, I demonstrated that it is possible to search for programs that have simple relationships between their numerical input and the number of clock cycles they consume, and that exhibit those relationships more accurately than "obvious" handwritten alternatives.

Consider now a more complex relationship such as a Boolean function. If we provide two Boolean inputs a and b, can we evolve a

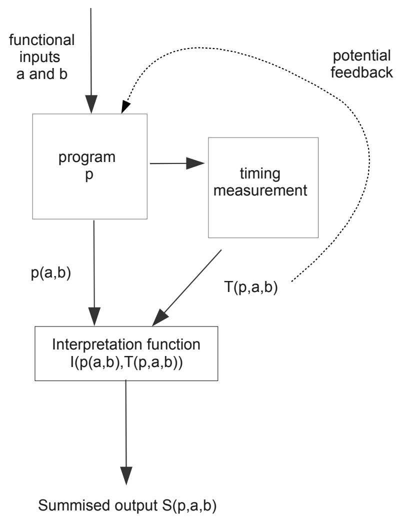


Figure 47: Visualising the use of timing in calculation.

program such that the cycle count of program p on those inputs $T(p, a, b) = f(a, b)$, where f could be (for example) a Boolean OR? For this to be literally true, the program's execution would have to take either 0 or 1 cycles, which is unrealistic as even a simple `if` will usually take more than a single cycle. Hence, we must find a suitable *interpretation* of the timing output, which I denote $I(T(p, a, b))$. This is an idea with much generality, and is illustrated in Figure 47.

A simple interpretation that can be employed here is to measure the number of cycles the machine has executed and then take modulo 2 of this number. Thus the lowest "bit" of the execution time is used as output, and there are two output states: an even and odd number of cycles consumed. An interpretation such as this is arbitrary, and the idea can be generalised: perhaps interpretations can be cooperatively coevolved. The most interesting question here is: do such code fragments even exist, given the context of the test program within which they will be used?

It is not obvious how to go about manually constructing a solution, but we can imagine two ways in which it may be achieved: through something we could describe as either *implicit* or *explicit* time variation. In the former, we hope to rely purely on the low-level mechanics of the processor and memory subsystem to provide variation, such as floating point operations that consume a variable number of cycles. In the latter, we rely on some logical test to choose a path through the code. In practice, the latter cannot succeed without the former in order to "iron out" variation as in the handwritten linear solution from Section 8.4.1.

8.5.1 *Experiment C: Timing OR Function*

In this experiment, I give as input the four possible combinations of a pair of Boolean-valued floats and try to evolve a program that outputs

Objective	Find a program $p^*(a,b)$ such that $T(p,a,b)\ mod\ 2 = OR(a,b)$
Terminal set	a, b, tmp, 1, Float_MAX
Function set	FixedLoop, $\leq$, increment tmp, $+$, $*$, $-$, $/$, if, sequence, skip, update tmp
Fitness cases	$a,b \in \{\{0,0\},\{0,1\},\{1,0\},\{1,1\}\}$ (repeated four times, order randomised at each generation)
Fitness function	$f(p) = w_1 \cdot c(p,a) + w_2 \cdot c(p,b) + w_3 \cdot (n - \sum_1^n \lvert T(p,a,b) - (a \vee b)\rvert)$
Parameters	Initial tree depth = 3, generations = 20, population size = 100, prob(xo) = 0.9, prob(mutation) = 0.1
Initialisation	Koza's ramped half-and-half method

Table 18: Experiment C: Settings to evolve a Boolean OR function using time as an output channel.

their OR on the lowest bit of the cycle count. Initial experimentation revealed the following:

- The test cases could be passed by code that did not even evaluate the inputs.
- Degenerate behaviours taking a single cycle ("always 1") were common.
- The ordering of test cases affected the performance of the search.

The code was therefore subsequently tested with the four possible inputs over four repetitions, and a randomised ordering of these 16 test cases used. In order to ensure that both inputs were evaluated at least once in any solution, the fitness function was specified as follows:

$$\begin{aligned} f(p) &= w_1 \cdot c(p,a) + w_2 \cdot c(p,b) \\ &+ w_3 \cdot (n - \sum_1^n \lvert I(T(p,a_n,b_n)) - (a_n \vee b_n)\rvert) \end{aligned} \tag{8.4}$$

where $c(p,a)$ is 1 if program p reads variable a and 0 otherwise, and $I(x) = x\ mod\ 2$. The parameters to the experiment are given in Table 18. Note the addition of the constants 1 and Float_MAX (3.40282e38) to the terminal set.

GP was indeed able to provide such a solution:

```
if (( 3.40282e+038f - b )) {
        for (c=0;c<(b + a );c++) {
                tmp++;
        };
} else {
        tmp =  3.40282e+038f;
}
```

This code passed all 16 test cases correctly, given the randomised order of tests at that generation. It is quite straightforward to see how this might work. However, running the code again with a different order of test cases resulted in the code failing two test cases. This alludes to the fact that T is a function not only of p, a and b, but also of the machine context ϕ at the time, so we are actually trying to evolve code such that $T(p, a, b, \phi) \; mod \; 2$ is equivalent to $a \vee b$. The context is the machine state: the content of the stack, the instruction pipeline, cache and so on. This reliance on context is both a difficulty, in the case where we wish to eliminate its effects entirely, and also a useful resource, in the case where we want to exploit the machine state to achieve certain timing properties. This context caused problems for the manual solution in Section 8.4, and is found to be useful in Section 8.6.

If we wish to implement this OR gate robustly, we must test on all $4! = 24$ possible input sequences. Even then, however, we cannot be sure that the context of the surrounding test program is not being exploited, such that if we wish to place the code in another program we would have to again evolve a solution and exhaustively test the possible outcomes. This context-sensitivity mean that any such "absolute time manipulation" must be done in situ.

I did indeed attempt to evolve a program that is robust to all input orderings, but failed to successfully do so even when exhaustively testing each bit of the timing output as a potential interpretation. Every run provided an individual with a few failed test cases out of the total 96. It may be that to achieve such control over absolute cycle usage it is necessary to carefully construct a function set that enables a variety of variations in timing.

8.6 TIMING AVALANCHE CRITERION

In Chapter 6, I demonstrated that it is possible to evolve light-weight, low-power pseudorandom number generators (PRNGs) using GP and simulation for evaluation. As a measure of the effectiveness of the PRNG, I used the Strict Avalanche Criterion (SAC) [Webster and Tavares, 1986], a method from cryptography that measures the nonlinearity of a given function. SAC analyses the expected distance between outputs given a single bit flip in the input. Please note that in this experiment I have used a single 32-bit input rather than 8 as in Chapter 6, for the sake of simplicity.

Each output bit should have a probability of 0.5 of being flipped when a single input bit is changed in order to maximise the nonlinearity of the PRNG. Hence the Hamming distance between the two outputs should follow the binomial distribution $B(n, \frac{1}{2})$. By recording the Hamming distance between $p(a)$ and $p(a')$ for each test case, a χ^2 squared goodness-of-fit measure can be calculated against the ideal binomial distribution of bit flips. The performance measure of an individual program p is given by:

$$SAC(p) = \sum_{i=0}^{n} \frac{(C_i - E_i)^2}{E_i} \tag{8.5}$$

C_i is the counted frequency of i bit flip events, and E_i the expected frequency. In this experiment, *I apply the avalanche criterion not to the functional output of the program but to its timing*, $T(a, p, \phi)$, a very different application of the SAC. This measure is henceforth referred to as the Timing Avalanche Criterion (TAC). This is a fascinating concept, which may never have been considered before simply due to the lack of any manual method capable of implementing it. It may enable us to evolve programs designed to be resistant to side-channel cryptanalysis.

8.6.1 *Side-Channels*

Embedded systems are particularly vulnerable to side-channel analysis, whereby an attacker exploits a non-computational channel of information to break the security of a system. Covert channels are also a generalisable concept [McHugh, 1995] that offer a mechanism not only for deriving information but also deliberately transmitting information about a system.

Timing channels constitute one specific class of covert channel. Physical timing properties can be used to transmit and derive information about an executing system. The leaking of information via side-channels is noted to be a major problem in designing secure algorithms and viewed somewhat pessimistically by security researchers:

> "It is probably not possible to protect against side-channel attacks in the design of algorithms." Kelsey et al. [1999]

Kocher [1996] was amongst the first to demonstrate that cryptographic primitives provably secure in the mathematical domain can become exposed to unseen vulnerabilities when implemented in a physical system. By monitoring the timing properties or power consumption [Kocher et al., 1999] of a system, it is possible to deduce information about the state of executing software and compromise its security. A wide variety of attacks exist, from simple timing of operations to statistical analysis of power traces.

Counteracting such attacks is difficult: one method is to design code to consume the same number of cycles regardless of both the data input or output and the state of the system. For example, this can be achieved through the insertion of NOP instructions through the implementation. The difficulty with this, as we saw previously, is that the complex interactions between software and machine state are not easily anticipated and may leak information regardless of our attempts to homogenise resource consumption. This defence is also usually vulnerable to other forms of attack, such as detecting NOPs through other means, and a more robust method of defence is desirable.

8.6.2 *Experiment D: Evolving TAC*

In this section, I attempt to evolve an expression that has a good TAC measure across the lowest 10 bits of its cycle count, which allows for up to 1023 cycles, sufficient for the evaluation of small expressions. Does such an expression even exist, given the context of a simple test loop that repeatedly uses the code with inputs separated by a single bit flip? It is not clear how we would proceed in designing such an expression, but can GP design one for us?

I use a sample size of 4000, which is generous according to previous analysis, and the same function set as in Chapter 6, with the addition of an "inline if" statement. This is added because it would seem intuitively

Objective	Find a program $p^*(a)$ such that $HD(T(p,a,\phi),T(p,a',\phi)) \approx B(n,\frac{1}{2})$ where $HD(a,a')=1$.
Terminal set	a, Integer ERCs
Function set	Inline if, <, LogicalShiftLeft (LSL), LSR, MULT, SUM, AND, NOT, OR, XOR
Fitness cases	a, a' where $HD(a,a')=1$, sample size 4000
Fitness function	$\sum_{i=0}^{n} \frac{(C_i-E_i)^2}{E_i}$ over $n=10$ bits of timing measurement, where C_i is the resulting frequency of i bits flipping and E_i is the expected frequency.
Parameters	Initial tree depth = 3, generations = 25, population size = 100, prob(xo) = 0.9, prob(mutation) = 0.1
Initialisation	Koza's ramped half-and-half method

Table 19: Experiment F: Settings to evolve an expression with a good Timing Avalanche Criterion measurement.

likely to be a useful function in achieving TAC. The experiment is summarised in Table 19. The best individual was subsequently tested over a sample size of 10000, which gave a good *TAC* measure of 0.0228. The *p* value of this result, effectively giving the probability that this sample is drawn from the ideal Binomial distribution, is 1.00 (to 3 s.f). The distribution of bit flips for this individual is illustrated in Figure 48. The individual is given below and is reasonably small. Note that I have not simplified this individual: I cannot even attempt to do so without potentially changing its timing behaviour!

```
tmp = (((a > (3594493887u)) ^ ((a * (1408948682u) ) >
    ((a > (3594493887u)) ^ (a * (609711807u) ) )) ) ?
    (((a * (1408948682u) ) > ((2302909662u) ^ a )) <<
    (((a > (a * (609711807u) )) ^ (a * (609711807u) ) )
     % 32u) ) : ((((2390510013u) * (((a > (3594493887u)
    ) ^ ((((a * (1408948682u) ) > (((a * (1408948682u)
    ) > ((2302909662u) ^ a )) << (((a > (a * (609711807
    u) )) ^ (a * (609711807u) ) ) % 32u) )) *
    (2540811676u) ) * (((a * (1408948682u) ) & ((a >
    (3594493887u)) ^ (a * (609711807u) ) )) + a ) ) ) &
     (2490185230u)) ) ^ (a * (609711807u) ) ) &
    (1135240832u)))
```

8.6.3 *Experiment E: TAC and SAC*

Building on the preceding result, I now ask: is it possible to produce programs that perform two tasks at once? That is, with both desirable functional and timing outputs? I attempt to produce code that has both good functional SAC (that is, it is a good random number generator by that criterion) and also good TAC (it is resistant to side-channel analysis). Such a solution provides a proof-of-concept for the ability of GP to evolve primitives resistant to side-channel analysis. It also opens up the interesting possibility of combining the timing and functional

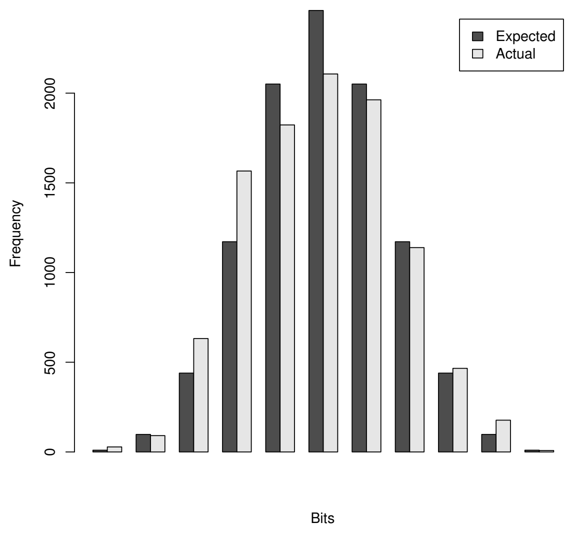


Figure 48: Bit flip distributions for simple TAC.

properties of the software to improve the output of the random number generator, or even to feedback the timing measure as input. This could be used to improve the nonlinearity of the code over a series of evaluations, assuming the capability to measure time from the code.

In order to evaluate both SAC and TAC, it is necessary to run the simulator twice. This is because outputting the functional results of the code requires the addition of a `printf` statement to the test harness, which changes the machine state and therefore the context of the code. Two test harnesses are used, and the individual is compiled and run through the simulator a second time to measure its functional behaviour. This is illustrated in Figure 49.

The fitness function used was an even weighted combination of functional avalanche criterion measure (SAC) and timing measure (TAC):

$$f(p) = w_0 \cdot \sum_{t=0}^{t_{max}} \frac{(C_t - E_t)^2}{E_t} + w_1 \cdot \sum_{s=0}^{s_{max}} \frac{(C_s - E_s)^2}{E_s} \tag{8.6}$$

where C_t and E_t are the actual versus expected bit flip frequencies for the timing channel output, and C_s and E_s those for the functional output. The frequencies were measured over $t_{max} = 10$ and $s_{max} = 32$ bits. The weightings $w_0 = w_1 = 0.5$ were chosen for this experiment. With a larger run of 25 generations, one of the best individuals produced was:

```
tmp = ((((a * (1408948682u) ) ^ ((((((((1106785805u) * a ) >>
    (((((1167812458u) + (2754164240u) ) & (~ a)) + ((2240102872u
    ) > a) ) % 32u) ) * ((((((1106785805u) * a ) >> ((a ^ a ) %
    32u) ) >> ((a ^ a ) % 32u) ) * ((((((2390510013u) * a ) >>
    (((1106785805u) * a ) % 32u) ) * (((1106785805u) * a ) >
    (2390510013u)) ) ^ (~ a) ) * a ) ) ^ (~ ((((1106785805u) * a
     ) > (3594493887u)) ^ a )) ) ) ^ ((1106785805u) * (~
    ((((1106785805u) * a ) > (3594493887u)) ^ a )) ) ) >> (a %
    32u) ) * (1408948682u) ) ^ ((((((2390510013u) * a ) >> ((a ^
     (((1876056559u) + (3922502476u) ) < ((1106785805u) * a ) )
```

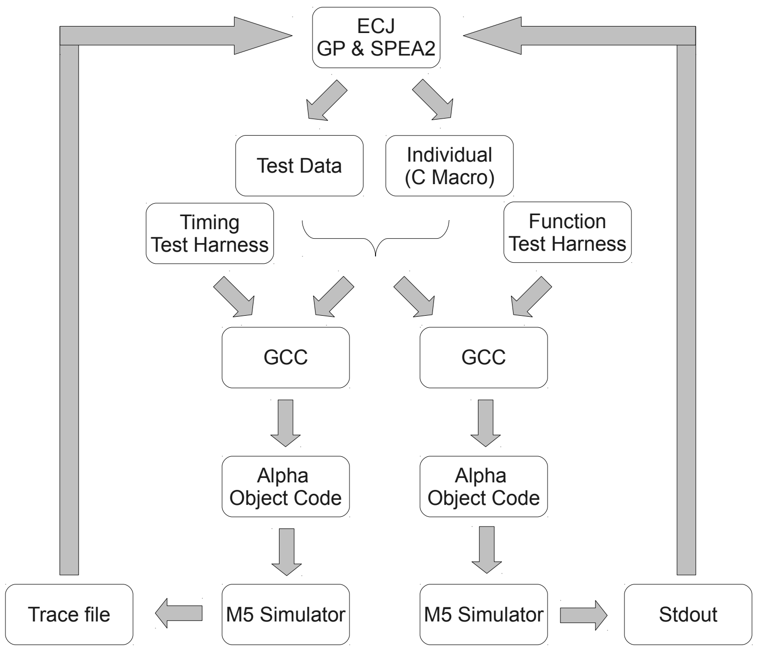


Figure 49: Evaluation of both timing and functional properties.

```
) % 32u) ) * (((2390510013u) * a ) >> ((a ^ ((1106785805u) *
 a ) ) % 32u) ) ) ^ (~ a) ) * a ) ) ) * (1408948682u) ) ^
(((((2390510013u) * a ) >> (((((((((a ^ (((2641736152u) *
(1408948682u) ) ^ ((3594493887u) * a ) ) ) >> ((a ^ a ) % 32
u) ) * ((((((2390510013u) * a ) >> (((1106785805u) * a ) %
32u) ) * (((1106785805u) * a ) > (2390510013u)) ) * a ) >>
((a ^ ((1106785805u) * a ) ) % 32u) ) ) ^ (~ a) ) >> ((a *
(609711807u) ) % 32u) ) * (((((1670273053u) | a ) &
((3825661740u) + (3105575741u) )) * ((((((2390510013u) * a )
 >> (((1106785805u) * a ) % 32u) ) * (((1106785805u) * a ) >
 (2390510013u)) ) ^ (~ a) ) * a ) ) ^ (~ ((((1106785805u) *
a ) > (3594493887u)) ^ a )) ) ) ^ ((1106785805u) * a ) ) >>
(a % 32u) ) % 32u) ) * (((2390510013u) * a ) >> ((a ^
((1106785805u) * a ) ) % 32u) ) ) ^ (~ a) ) )
```

Comparing this to results in Chapter 6, there is a great deal more use of constants. Constants require memory access, which introduces variation into the timing properties of the individual. The TAC and SAC distributions for this individual over a larger sample of 10 000 are given in Figure 50. Over that sample, this individual had a SAC of 0.0189 and a TAC of 0.0399. Both are equivalent to a p value of 1.00 (to 3 s.f.). These are excellent values, achieved surprisingly easily by GP despite having to satisfy two requirements simultaneously. Table 20 gives a breakdown of the usage of each function in the function set for the best individual from Chapter 6, the TAC individual from Section 8.6.2 and the best individual with both TAC and SAC.

Note that the first figures are for an individual taking 8 inputs, whereas the following two are only operating on a single input. The TAC individual is quite similar to the original individual, except that it employs more comparisons, which certainly adds timing variation. In summary, it appears that achieving TAC alone is not much more difficult than achieving SAC. The reason this might seem surprising to us is that we have an intuition for the behaviour of functional operators, but not for the timing of those operations and their interactions with state at a machine level. In contrast, GP measures the utility of an individual purely on "black box" feedback, and the origin of such information is of no concern to the search algorithm. As if to underline the weakness of our intuition, GP has used only a single if statement across the two individuals.

When evolving an individual with both TAC and SAC, the frequencies of functions used become much greater and a larger individual is produced. This does not appear to be the emergence of bloat (as a steady increase in fitness is observed), rather the size of the individual is a required property to achieve the goal. We must pay in size for the security that TAC may provide. Most operations increase in approximately linear proportion. The most striking differences are the use of shift operations, and subsequent requirement for modulo operators to prevent program errors. These appear to add much variation to the timing of the individual.

Overall, Table 20 summarises well the observations made in Chapter 3 regarding the different and competing goals in embedded systems programming. Firstly, we have three problems with different requirements that make them distinct from each other. Secondly, good solutions to each problem require different compositions of their solutions. Thirdly, Genetic Programming is able to easily adapt to the problem at hand: given different problems with the same function set and similar parameters, it is able to identify those operations that are most effective at achieving the desired functional and non-functional behaviour.

8.6.4 *Robustness*

The TAC of the individual found above is impressive, but was measured when tested only from a test harness, where the function was repeatedly called with hardcoded data from an array. What about its use in a deployed program? We could repeat the evolutionary search in the context of the system it is to be deployed in, which would require the construction of a suitable test harness that mimics the behaviour of the system to be deployed. It would be more convenient though if the search did not have to be repeated. Given the results in Section 8.5.1, we might be pessimistic about the ability of a code fragment to be robust enough to have similar properties when called in a very different context.

To test the robustness of the individual, I used it in the context of a randomised quicksort algorithm [Motwani and Raghavan, 1995]. Randomised quicksort selects the next pivot stochastically, which improves the expected time for the sort. A new test harness was created, which populates a large data array with random integers and then calls randomised quicksort to sort the array. At each pivot selection point, the individual was called to function as a random number generator and the input to the individual was taken as before from a hardcoded array of test data. The quicksort implementation used extensive recursion, such that the cache and stack contents are continuously changing through the different calls to the individual. A large enough array was used to ensure a sample size of 10000 calls to the individual, and the TAC calculated.

Function	Best from Chapter 6	TAC Only	TAC and SAC
Input values	20	21	56
Constants	3	24	61
SUM	4	1	4
MULT	7	15	44
NOT	2	0	9
XOR	10	9	24
AND	0	3	2
OR	0	0	1
Modulo	0	2	15
Less Than	0	0	1
Greater Than	0	10	7
Shift Right	1	0	15
Shift Left	0	2	0
If	0	1	0
Total	47	88	239

Table 20: Comparing the composition of best individuals from Chapter 6, TAC only problem and TAC-SAC problem, by the frequencies of each operation in the individual.

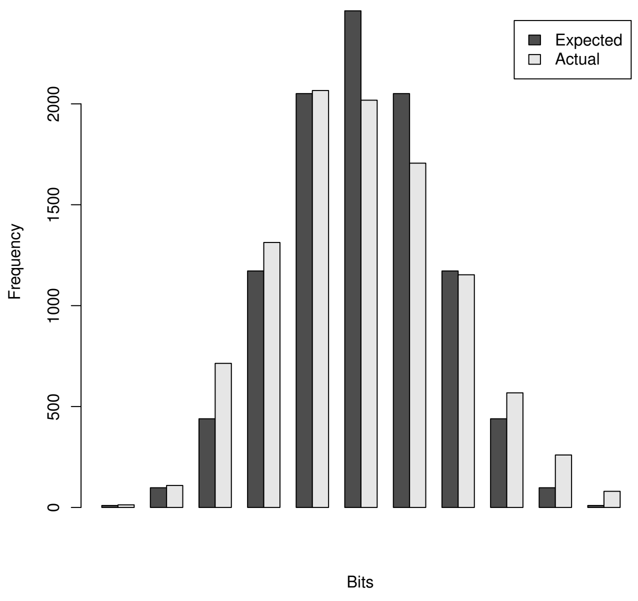
Surprisingly, despite the varying machine state that the function was called in, the TAC measure over 10000 samples was 0.180. This still gives a p value of 1.00 (to 3.s.f). The corresponding plot is given in Figure 51. Thus, the *individual's resistance to side-channel analysis* is a little degraded, but still an exceptionally good fit to the binomial distribution. This is a very important result, as it demonstrates the feasibility of evolving individuals that having statistical timing properties generalising between contexts.

8.7 SUMMARY

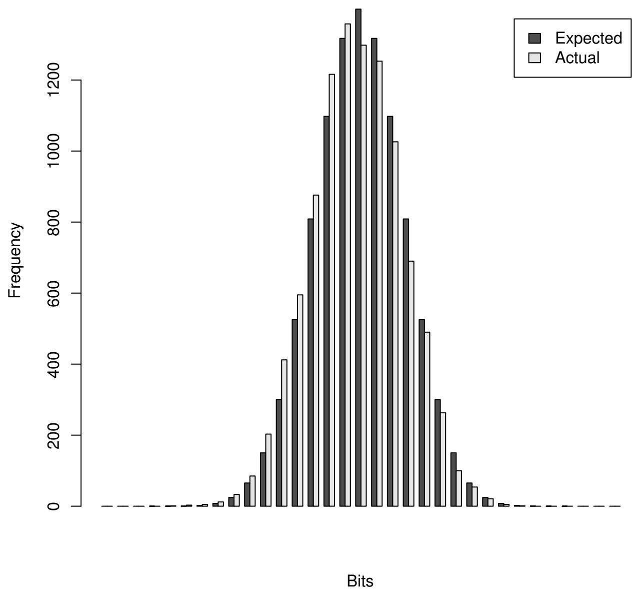
In this chapter I have introduced several ideas regarding the application of Genetic Programming in manipulating the fine-grained resource consumption of code. GP was shown to be most useful in controlling the behaviour of code over a series of evaluations, rather than the absolute value of a single evaluation. Not only was GP successful at producing desired statistical timing behaviour whilst simultaneously carrying out functional computation, but the individual also generalised over different calling contexts.

Such control opens up a whole avenue of further exploration in the concept of "doing two things at once", that is to liberate software from its role as an object of abstract calculation into a process that interacts with the host hardware platform and its execution context as part of its functional calculation. At such a complex level of interaction, methods such as Genetic Programming may be essential to achieve desired behaviour.

Example applications of these methods can be found in cryptography, both in defending against side-channel attacks and exploiting side-channel properties. By exploiting covert timing channels [Kemmerer,



(a) TAC



(b) SAC

Figure 50: Bit flip distributions for best TAC-SAC individual.

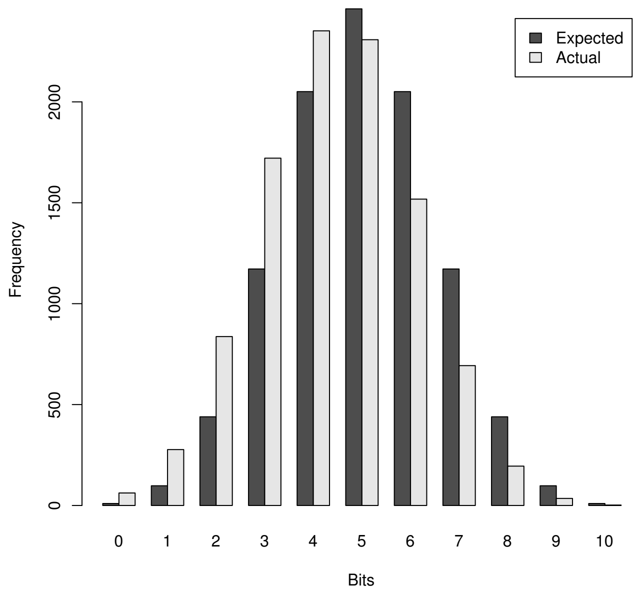


Figure 51: Bit flip distributions for best TAC-SAC individual when used in randomised quicksort.

2002], programs such as that evolved in Section 8.6.3 can be used to transfer information such that a *program is a key*, and the timing output of that program on a given input reveals the message. Only a party with the same code and an equivalent platform can decode it. Another scenario is using such a program as a keystream generator, incorporating the timing properties of the software by (for example) XORing the functional output with the time taken to generate the previous output.

This open-ended research concludes the empirical chapters. What follows is an evaluative discussion of how the empirical work presented supports the hypotheses set out at the start of the thesis, as well as the opportunities for further application and development that some of these experimental results present.

Part IV

CONCLUSION

9 EVALUATION & FUTURE WORK

9.1 INTRODUCTION

In this thesis, I set out three hypotheses regarding the application of Genetic Programming to produce software for low-resource systems, and in the preceding three chapters I have detailed empirical experimentation that serves to support these hypotheses. I have presented three different applications of GP to developing software for low-resource systems: to create entirely new software taking into account trade-offs between functionality and resource consumption, to improve existing software in terms of a non-functional quality of that software, and finally to create software with specific side-channel behaviour that goes beyond a single measurement of its non-functional properties.

In this chapter, I first return to the individual hypotheses and demonstrate how the work completed and described in this thesis supports them. After discussing the hypotheses, I emphasise the original contributions of the thesis, and also the results that exceeded or contradicted my expectations.

The strands of work completed form a wide front of research that could be pushed forward both in terms of its *sophistication* and also its *application*. I therefore outline some of both in this chapter, as a potential research agenda to carry this work forward to full maturity. I conclude with a summary of the insights this work has provided.

9.2 H1: CAPABILITY OF GP TO MAKE TRADE-OFFS

> **Hypothesis 1**: *Genetic Programming will be able to provide graceful degradation in the trade-off between resource consumption and functionality.*

In Chapter 6, Genetic Programming found a wide-range of trade-offs in objective space, where the objectives were power consumption and the functional quality of pseudorandom number generators (PRNGs). Graphs such as Figure 18 demonstrate how continuous this trade-off is: a fine level of granularity exists in the balance between the two objectives. The range of power consumption in this figure stretches from that required to simply return a constant, up to a program that provides an almost perfect functional fitness that uses much more energy.

Results in Chapter 7 also confirm the ability of the search algorithm to discover continuous surfaces in these types of objective spaces. In that work, the objectives were discrete-valued in contrast to the (theoretically) continuously valued objectives in the PRNG work. The ultimate goal in optimising the case studies in that chapter was to produce improved programs free of error, but nevertheless a continuous trade-off was found in all cases.

A practitioner can use GP to provide degradation as available resources decrease. Such degradation is useful in situations where sub-optimal functionality is better than none at all. A trade-off may be

Problem	GCC Inst. Count	GP Median Inst. Count
Triangle1	22402	17214.5
Triangle2	8380	8000
Remainder	27025	23318
Sort1	884842	569517
Sort2	815882	790062
Factorial	395189	278700
Switch 10	143000	36000
Select	97077	90201.5

Table 21: GCC-optimised code performance versus best individuals found using GP.

chosen statically at design-time or degradation provided at run-time by switching between different versions of a set of software provided by GP. I have already discussed mode switching in the context of power management in embedded systems in Section 3.3.5. Using the results of a GP search, power management could be applied to the whole *software ecosystem* as well as hardware modes. A system library may provide a set of implementations that vary in their power consumption, and switch to low-power implementations when the operating system detects low battery power. Thus the level of system service may gracefully degrade in the face of falling power supply.

9.3 H2: IMPROVING EXISTING SOFTWARE USING GP

> **Hypothesis 2**: *Genetic Programming will be able to optimise non-functional properties of software to a level not achievable by a compiler: in particular, solutions found by GP will Pareto-dominate hand-written solutions optimised by a compiler such as GCC.*

The investigation of this hypothesis lies solely in Chapter 7. This hypothesis is directly quantifiable, and Table 21 gives a convenient summary of results from that work. The table lists the median instruction counts across a range of parameter settings for best individuals produced by GP. These are compared to those produced by a GCC cross-compiler using the "-O2" optimisation level, the most efficient level that can be similarly traced using the simulator. Median counts are given to underscore the robustness of the method with respect to its parameters. The framework successfully optimised the programs thousands of times with different parameter settings.

A note of caution must be added here. The evolutionary optimisation used ran in time periods orders of magnitude longer than a compiler. Indeed, during evaluation the compiler itself was repeatedly executed.

How well a compiler author may do with a remit of finding optimisations that outperform GP in a similar timespan is not of immediate practical importance. It is actually the case that Genetic Programming and the compiler *are working together* to optimise the software. GP is in effect a sophisticated preprocessor that presents code in a form that enables the compiler to produce a more efficient implementation than otherwise possible. The two are in cahoots. Thus, any improvement in compiler technology may be incorporated within the evolutionary search. The flexibility of search algorithms in incorporating existing domain knowledge and toolsets in this way is one of their most appealing features.

9.4 H3: FINE-GRAINED CONTROL OF RESOURCE CONSUMPTION

> **Hypothesis 3**: *Genetic Programming can act as a mechanism to improve fine-grained control over emergent properties arising from the interaction between compiled source code and the host hardware platform by treating this system as a "black box", and discovering complex relationships through exploration of the search landscape.*

This hypothesis is the subject of Chapter 8. In that chapter, I demonstrated that GP could find software with timing properties that were non-trivial to design by hand. Furthermore, I was able to create software with desirable timing behaviour, where previously it was unknown if such software even existed.

Firstly, I illustrated the difficulty of producing code that has specific timing behaviour. Writing software to precisely exhibit simple linear and quadratic relationships between input and time consumption was difficult given a non-trivial machine and corresponding state. I then outlined an even more difficult task: producing software that consumes a quantity of time T, where T may be regarded as a random variable and its behaviour as such is quantified by a statistical test.

In contrast to the difficulty such a task may provide for a human programmer, the computational effort (number of fitness evaluations) that Genetic Programming required was surprisingly small. GP was able to evolve code with a good Timing Avalanche Criterion (TAC) measure. This may represent a new solution method to the problem of protecting against side-channel attack in cryptography. Not only that, but I was also able to demonstrate that this property could be maintained whilst also performing useful functional behaviour.

Further investigation demonstrated that software created using GP to exhibit this complex timing behaviour was robust to its execution environment. An individual evolved in a given test context maintained those properties when embedded in a randomised quicksort implementation.

These experiments provide evidence that GP can be used to control timing properties to an extent conventionally viewed to be infeasible.

9.5 THESIS HYPOTHESIS

The fundamental hypothesis of this research was:

> **Thesis Hypothesis**: *Program search is a versatile and effective method that can be used to satisfy the conflicting requirements of low-resource systems.*

The subhypotheses lend weight to this overarching one. Certainly, GP can explore trade-offs between requirements and provide superior solutions to those found in past research or using a modern compiler. Some of these results could be used immediately in an embedded system. The work in Chapter 8 is more explorative, yet promising. Developing these methods into a usable tool appears feasible, and it should also be possible to create reusable knowledge from results produced by these methods.

9.6 ORIGINAL CONTRIBUTIONS

The original contributions of this thesis are as follows:

- The combination of multi-objective optimisation, hardware simulation and evolutionary search to systematically optimise non-functional properties of software.

  Previously, multi-objective Genetic Programming has been used to a limited extent in order to optimise non-functional properties such as program size and robustness. However, these methods have not used hardware simulation to estimate and optimise dynamic properties such as execution time or power consumption. The methods presented here are both systematic and generalisable, in that they may be applied in the same way to other problems, properties or target platforms.

- Introducing GP as a decision-making tool where trade-offs between functional and non-functional objectives exist.

  The Pareto fronts in Chapter 6 give the engineer an insight into the nature of the trade-offs involved, and present a set of solutions to be chosen from. The same chapter also demonstrates the impact of including or omitting an operator (multiply) in terms of its impact on the trade-offs that can be achieved.

- The use of Genetic Programming along with other evolutionary methods to describe and approach in a systematic and generalisable manner the improvement of existing software.

  The combination of coevolution, seeding and multi-objective optimisation to *improve* non-functional properties rather than *create* efficient software using GP is unique to the best of my knowledge.

  This work suggests a broad frontier of problems that could be solved with such an approach. For example, a subject of recent development is bug-fixing existing software using Genetic Programming [Forrest et al., 2009]. This work focuses on functional repair alone, but here I have already demonstrated that existing software can be improved in a non-functional sense whilst retaining its functionality.

  The results obtained using this method are considerably better than using an optimising compiler alone. Some of the optimisations of Chapter 7 were very much unforeseen. The elegant use of an input variable as a loop counter to reduce execution time is a good example. Also, the ruthless nature of GP in removing unnecessary instructions and in particular exploiting the expected input distribution in order to do so, was quite impressive.

- Proposing, and demonstrating the feasibility of, search as a tool to enable fine-grained control of resource consumption.

  I have shown that GP can be used to control the relationship between a program's numerical input and its execution time in a superior way to "obvious" handwritten solutions. GP was able to control individual cycle consumption to the extent of calculating a Boolean function in terms of its timing, although not in a completely robust manner.

- The use of GP to successfully produce software with a novel timing property that has not been previously developed.

  My work shows that it is possible to evolve software that robustly exhibits a sophisticated statistical behaviour in its timing characteristics. This idea and the results produced are likely to be useful in security applications defending against side-channel attacks, by reducing information leakage. It wasn't clear whether such software would indeed exist per se. However, I have shown that it does, and the fact that it was found so quickly and consistently by Genetic Programming was both surprising and encouraging in the promise it holds for other applications.

9.7 refining the approach

Having discussed some of the achievements of the thesis, I now outline a way forward for future research in terms of developing further the methods devised and tested.

9.7.1 *Creating Scalable Solutions*

In the work presented, the output of the evolutionary run was often a *set of solutions* to be chosen from by a practitioner. The choice of trade-off is then statically determined, and a deployed system is limited in its capability to dynamically respond to resource availability. An alternative approach would be to create a single program that can adaptively trade-off between functionality and resource consumption. Hence a variable quality-of-service could be provided. This is similar to the concept of evolving anytime algorithms [Zilberstein, 1996]. Hand-crafting anytime algorithms with relatively simple and intuitive requirements has proved to be a difficult challenge for human programmers. Finding solutions that trade-off complex qualities such as power consumption in an anytime manner is likely to be a very difficult problem. However, once found, such artefacts would be reusable across different systems.

9.7.2 *Handling Expensive Fitness Functions*

The most immediate barrier to deployment is the length of time taken to execute an evolutionary run. Available simulators remain slow, for example evaluations from Chapter 8 took as long as 10 minutes to evaluate functional and non-functional behaviour for a single individual. In Chapter 7 I demonstrated the use of modelling as an alternative to full simulation, and more sophisticated methods from embedded and real-time systems research could be adopted in a similar vein.

Figure 52 summarises the different levels of fidelity that can be used to evaluate a solution during the development of an embedded system. In this thesis I have applied modelling, interpretation and variable levels of detailed simulation. Note that whilst external hardware may be faster than simulation for a single run, in fact it is to be expected that evaluation using external hardware will take longer than simulation.

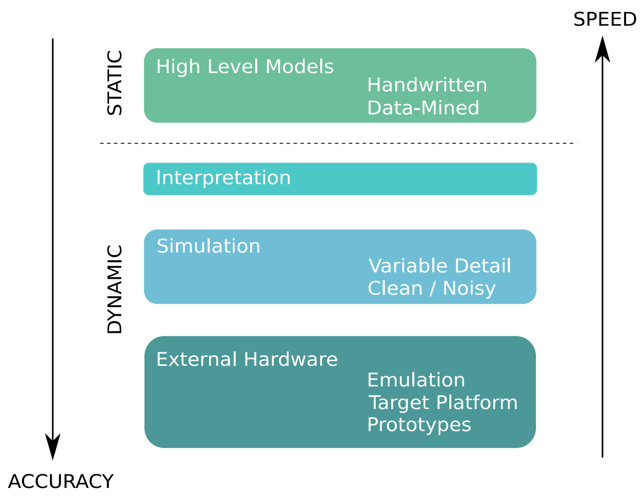


Figure 52: Variable fidelity in measuring the behaviour of embedded software.

This is because of the difficulties of accurate general measurement of non-functional properties, a problem which must be compensated for with multiple executions of a single test case to eliminate the impact of noise.

Observing that (a) often evaluation need only give relative measures of individuals' fitnesses and that (b) modelling has shown to be a useful approximation of simulation, it makes sense to propose that a *hybrid evaluation method* is employed. There is much to answer here, for example:

- When should we employ hardware-based evaluation and when should we use lower fidelity measures?
- How can we efficiently allocate evaluations across different methods? For example, can we use external hardware to validate previous estimations of behaviour whilst continuing in parallel to use software estimation to continue the search?
- After obtaining the results of evaluation at a given level of fidelity, how best to use that information to automatically improve the quality of higher-level methods? Can we feed information obtained upwards through this hierarchy?

9.8 SEMANTIC CORRECTNESS

A drawback of using Genetic Programming to search for programs is that fitness evaluation is usually measured by a set of test cases. Any behaviour that is not explicitly verified by the test set cannot be guaranteed, i.e. we cannot be sure that a solution will generalise to other inputs. In the case where there is a non-Boolean acceptability of functionality, this does not pose a problem. The PRNGs produced in Chapter 6 are an example of such a situation. If, however, a solution has a Boolean measure of acceptability, as in Chapter 7, then we face a dilemma. We may accept a result based on further examination using a validation set, in the hope that this will prove sufficient. This is not always unreasonable – we rely on hand-written software that has been subjected to similar testing, although some argue that a human author may write more reliable code due to their experience and understanding of a problem. An alternative approach is to manually verify the results produced by GP, as I did with some examples in Chapter 7.

If we wish to automate as much of the optimisation process as possible, it would be preferable to use a technique such as model-checking to validate our output, as suggested by Johnson [2007]. Model-checking has the additional benefit of being able to reason over concurrent systems, and in some cases resource usage can be modelled [Norman et al., 2005].

9.9 FURTHER APPLICATIONS

In this section, I outline a number of applications for the methods proposed in this thesis. These include applications that could be implemented without further refinement, and more speculative ideas that require investigative experimentation.

9.9.1 *Operating System Applications*

Operating System components are suitable targets for evolutionary methods, as they provide self-contained functions that are small in size (with a correspondingly small search space) and are frequently used. Small improvements in power efficiency, for example, make a significant contribution to the overall efficiency of the system.

Interrupt handlers are one such example. What they must achieve is usually very well-defined and of limited complexity, and it is of utmost importance that handlers exit quickly to avoid interrupting other processes more than necessary. Methods such as those outlined in Chapter 7 could, in combination with search algorithms like Linear GP, be applied to this domain. Such components have a Boolean measure of acceptability, but they are also small enough to verify manually and are a good candidate for deriving optimisations from GP output.

Another example is the problem of context-switching and other kernel-based housekeeping, which constitute a large timing and power consumption overhead [Tan et al., 2005]. These are small operations with a precisely defined result that may benefit from fine-tuning using evolutionary search to locate novel optimisations. These applications are most likely to find success in systems employing CISC architectures, where subtle interactions within the target platform may provide opportunities for fine-grained improvement.

9.9.2 *Improving Compilers*

Chapter 7 demonstrates the capability of GP-based search to find optimisations that modern compilers cannot. In that application, the benefits of GP were additive to those of the compiler. However, there are also many tasks that a compiler must perform that are candidates for efficiency improvement using GP. For example, embedded systems may employ 8-bit or 16-bit processors rather than the 32-bit or 64-bit architectures used in desktop PCs. Code that assumes 32-bit capabilities may be compiled for a target platform without 32-bit instructions, requiring conversion to simpler sequences that can be executed on that platform. Discovering the most efficient sequences for individual platforms is a potential application of GP.

Similarly, compilers incorporate standard patterns of transformations known to improve efficiency (loop-unrolling, inlining etc.). In my work, GP has found some optimisations that appear to fit into such a standard repertoire. Perhaps by exploring large amounts of optimisation data found by Genetic Programming using data mining methods, common patterns could be abstracted from that data and incorporated statically into compilers. A similar application of searching for the optimal sequence of peephole optimisations was investigated by Wild [2002].

9.9.3 *Exploiting Side-Channel Information*

In Chapter 8 I demonstrated that it is possible to achieve very fine-grained control over properties that are an emergent phenomenon of the software design, compilation and hardware implementation process.

In one particular application, the aim was to provide resistance to security attacks, but the idea has much wider potential. Given GP as a tool to exercise such control, we can consider non-functional behaviour as a secondary output channel. Can we use the physical properties of software-hardware interaction to improve our computation? We may effectively "do two things at once", leading to smaller, faster or more power-efficient solutions than possible without using these channels.

Consider a program that generates a sequence. What if we could find a solution that outputs the nth element of the sequence functionally, and the $n + 1$th on a non-functional channel simultaneously? Or, by feeding back timing measurement into a PRNG, could we produce a more compact solution? Similarly, we could consider steganographic applications. Reverse engineering using GP to clone a system could incorporate non-functional properties as channels of information to aid the search.

9.10 final words

The empirical experimentation presented in this thesis provides evidence in support of the hypotheses set out in Chapter 2. This experimentation may serve as a foundation for exploitation and further exploration of these methods, which lie on a broad front across evolutionary computation, simulation-based evaluation and compiler and embedded systems design. The methods developed may enable us to greatly improve our control over the non-functional properties of software, an essential capability as we develop new systems that push the boundaries of physical resources and blur the lines between hardware, software and the physicality of computation.

BIBLIOGRAPHY

Alien technology. URL http://www.alientechnology.com. (Cited on pages xiv and 3.)

Crossbow Technology. URL http://www.xbow.com. (Cited on pages xiv and 3.)

Xilinx, inc. URL http://www.xilinx.com. (Cited on pages xiv and 3.)

A. Acquaviva, T. Simunic, V. Deolalikar, and S. Roy. Remote power control of wireless network interfaces. *Journal of Embedded Computing*, 1(3):381–389, 2005. (Cited on page 17.)

W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Softw. Technol.*, 51 (6):957–976, 2009. (Cited on page 5.)

A. Agapitos and S. M. Lucas. Evolving efficient recursive sorting algorithms. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 9227–9234. IEEE Press, 2006. (Cited on page 30.)

A. H. Aguirre, R. S. Zebulum, and C. Coello Coello. Evolvable hardware. In *6th NASA / DoD Workshop on Evolvable Hardware*, pages 199–205. IEEE Computer Society, 2004. (Cited on page 56.)

A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977. ISBN 0201000229. (Cited on pages 21 and 23.)

I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002. (Cited on pages 21 and 69.)

W.G. Alexander and D.B. Wortman. Static and dynamic characteristics of xpl programs. *Computer*, 8(11):41–46, 1975. (Cited on page 14.)

J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006. (Cited on page 90.)

P. J. Angeline. Genetic programming and emergent intelligence. In *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994. (Cited on page 49.)

P. J. Angeline. Comparing subtree crossover with macromutation. In *EP '97: Proceedings of the 6th International Conference on Evolutionary Programming*, pages 101–112. Springer-Verlag, 1997. (Cited on page 52.)

P. J. Angeline. Competitive fitness evaluation. In *Evolutionary Computation 2: Advanced Algorithms and Operators*, chapter 3, pages 12–14. Institute of Physics Publishing, Bristol, 2000. (Cited on page 65.)

P. J. Angeline and Jordan Pollack. Evolutionary module acquisition. In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, 1993. (Cited on page 47.)

A. Arcuri, D. R. White, J. A. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning (SEAL)*, pages 61–70, 2008. (Cited on pages vii and 85.)

R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 73–78, 2002. (Cited on page 16.)

W. Banzhaf, P. Nordin, R. E. Keller, and F. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998. ISBN 155860510X. (Cited on pages 29, 37, and 39.)

B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold, 1990. ISBN 0442206720. (Cited on page 85.)

L. Benini, A. Bogliolo, and G. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, 2000. (Cited on page 17.)

L. Benini, A. Macii, and M. Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Transactions on Embedded Computing Systems*, 2(1): 5–32, 2003. (Cited on page 15.)

R. Berntsson Svensson, T. Gorschek, and B. Regnell. Quality requirements in practice: An interview study in requirements engineering for embedded systems. In *REFSQ '09: Proceedings of the 15th International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 218–232. Springer-Verlag, 2009. (Cited on pages 4 and 57.)

M. Berthold and D. J. Hand. *Intelligent Data Analysis: An Introduction*. Springer-Verlag, 1999. ISBN 3540658084. (Cited on page 75.)

N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006. (Cited on pages 86, 92, and 125.)

S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: Reducing bloat using spea2. In *Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001)*, pages 536–543. IEEE Press, 2001. (Cited on pages 50 and 66.)

B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, 2005. Springer. ISBN 3540251073. (Cited on page 18.)

M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007. ISBN 0387310290. (Cited on page 47.)

D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000. (Cited on page 71.)

D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, Computer Sciences Department. University of Wisconsin-Madison, 1996. (Cited on page 71.)

E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: an emerging direction in modern search technology. In *Handbook of metaheuristics*, pages pp. 457–474. Kluwer Academic Publishers, 2003. (Cited on page 30.)

D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, pages 40–52, 1991. (Cited on page 16.)

L. Carro, M. Kreutz, F. R. Wagner, and M. Oyamada. System synthesis for multiprocessor embedded applications. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 697–702. ACM Press, 2000. (Cited on page 23.)

A. Charnes, W. W. Cooper, and R.O. Ferguson. Optimal estimation of executive compensation by linear programming. *Management Science*, 1(2):138–151, 1955. (Cited on page 58.)

H. Chen, J. A. Clark, and J. Jacob. Automated design of security protocols. *Computational Intelligence*, 20(3):503–516, 2004. (Cited on page 50.)

J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, K. Rees, and M. Roper. Reformulating software engineering as a search problem. *IEE Proceedings – Software*, 150, 2003. (Cited on page 5.)

L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22:471–489, 2000. (Cited on page 13.)

C. Coello Coello. An updated survey of GA-based multiobjective optimization techniques. *ACM Computing Surveys*, 32(2):109–143, 2000. (Cited on page 55.)

C. Coello Coello, D. Van Veldhuizen, and G. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer Academic Publishers, 2002. ISBN 978-0-387-33254-3. (Cited on page 55.)

W. Comisky, J. Yu, and J. R. Koza. Automatic synthesis of a wire antenna using genetic programming. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, pages 179–186, 2000. (Cited on page 31.)

K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the workshop on languages, compilers, and tools for embedded systems*, pages 1–9, 1999. (Cited on page 26.)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2001. (Cited on page 96.)

S. L. Coumeri and D. E. Thomas. Memory modeling for system synthesis. *IEEE Transactions on VLSI Systems*, 8(3):327–334, 2000. (Cited on page 15.)

N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187. Lawrence Erlbaum Associates, Inc., 1985. (Cited on page 29.)

I. Das and J. Dennis. A closer look at drawbacks of minimizing weighted sums of objectives for pareto set generation in multicriteria optimization problems. *Structural and Multidisciplinary Optimisation*, 14(1): 63–69, 1996. (Cited on page 58.)

K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001. ISBN 047187339X. (Cited on pages 5, 55, and 62.)

R. P. Dick and N. K. Jha. Mogac: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 522–529. IEEE Computer Society, 1997. (Cited on page 56.)

S. Dignum and R. Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595. ACM Press, 2007. (Cited on page 50.)

W. E. Dougherty, D. J. Pursley, and D. E. Thomas. Instruction subsetting: Trading power for programmability. In *Proceedings of the Computer Society Workshop on VLSI System-Level Design*, 1998. (Cited on page 15.)

M. Ebner. On the search space of genetic programming and its relation to nature's search space. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages –1361, 1999. (Cited on page 32.)

ECJ. Evolutionary Computation in Java, 2009. URL `http://www.cs.gmu.edu/~eclab/projects/ecj/`. (Cited on pages 72, 97, and 125.)

A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003. ISBN 3540401849. (Cited on page 38.)

ENT. A Pseudorandom Number Sequence Test Program, 2009. URL `http://www.fourmilab.ch/random/`. (Cited on page 82.)

R. Feldt and P. Nordin. Using factorial experiments to evaluate the effect of genetic programming parameters. In *Proceedings of the European Conference on Genetic Programming*, pages 271–282. Springer-Verlag, 2000. (Cited on page 100.)

C. Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. *ArXiv Computer Science e-prints*, 13:87–129, 2001. (Cited on page 47.)

D. B. Fogel and J. W. Atmar. Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63(2):111–114, 1990. (Cited on pages 32 and 52.)

C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423. Morgan Kaufmann, 1993. (Cited on page 62.)

R. Forré. The strict avalanche criterion: spectral properties of boolean functions and an extended definition. In *CRYPTO '88: Proceedings on Advances in cryptology*, pages 450–468. Springer-Verlag, 1990. (Cited on page 73.)

S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009. (Cited on page 148.)

B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. *SIGPLAN Not.*, 40(7): 78–86, 2005. (Cited on page 25.)

A. A. Freitas. A critical review of multi-objective optimization in data mining: a position paper. *SIGKDD Explor. Newsl.*, 6(2):77–86, 2004. (Cited on page 55.)

G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, pages 1–13, 2008. (Cited on page 26.)

J. G. Ganssle. *Art of Designing Embedded Systems*. Butterworth-Heinemann, 1999. ISBN 0750698691. (Cited on pages 13 and 25.)

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989. ISBN 0201157675. (Cited on pages 38 and 42.)

C. P. Gomes and B. Selman. Practical aspects of algorithm portfolio design. In *Proceedings of the Third ILOG International Users Meeting*, 1997. (Cited on page 23.)

D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147. ACM, 2003. (Cited on page 15.)

B. Gorjiara, M. Reshadi, and D. Gajski. Designing a custom architecture for dct using nisc technology. In *Design Automation, 2006. Asia and South Pacific Conference on*, 2006. (Cited on page 26.)

Y. Y. Haimes, L. S. Lasdon, and D. A. Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *Systems, Man and Cybernetics, IEEE Transactions on*, 1 (3):296–297, July 1971. (Cited on page 59.)

Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007. (Cited on page 5.)

M. L. Harrison and J. A. Foster. Evolvable hardware. In *6th NASA / DoD Workshop on Evolvable Hardware (EH 2004)*, pages 123–129. IEEE Computer Society, 2004. (Cited on page 26.)

N. Hatta, N. D. Barli, C. Iwama, L. D. Hung, D. Tashiro, S. Sakai, and H. Tanaka. Bus serialization for reducing power consumption. *IPSJ Digital Courier*, 2:165–173, 2006. (Cited on page 16.)

T. Haynes, R. L. Wainwright, S. Sen, and D. A. Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann, 1995. (Cited on page 51.)

S. Heath. *Embedded Systems Design*. Butterworth-Heinemann, 1997. ISBN 0750632372. (Cited on pages 3, 13, 15, 18, 24, and 25.)

Thomas A. Henzinger and Joseph Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007. (Cited on page 4.)

J. C. Hernandez, P. Isasi, and A. Seznec. On the design of state-of-the-art pseudorandom number generators by means of genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1510–1516, 2004. (Cited on pages 70, 71, and 82.)

W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990. (Cited on pages 65 and 91.)

K. Holladay, K. Robbins, and J. Ronne. Eurogp. In *Proceedings of the 10th European Conference on Genetic Programming, EuroGP 2007*, volume 4445, pages 102–113. Springer, 2007. (Cited on page 48.)

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, 1975. (Cited on pages 29, 32, and 42.)

K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 165–174, 2008. (Cited on page 26.)

W-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Thermal-aware task allocation and scheduling for embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 898–899. IEEE Computer Society, 2005. (Cited on page 19.)

Huygens Search and Optimisation Benchmarking Suite, 2007. URL `http://gungurru.csse.uwa.edu.au/cara/huygens/`. (Cited on pages xiii and 37.)

Intel Atom Processor, 2009. URL `http://www.intel.com/technology/atom/`. (Cited on page 14.)

C. Isen, L. K. John, and E. John. A tale of two processors: Revisiting the risc-cisc debate. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 57–76. Springer-Verlag, 2009. (Cited on page 14.)

M. F. Jacome and A. Ramachandran. *Embedded Systems Handbook*, chapter 16 (Power-Aware Embedded Computing). Taylor and Francis, 2006. ISBN 0849328241. (Cited on pages 4, 13, 19, and 93.)

J. Jannink. Cracking and co-evolving randomizers. *Advances in Genetic Programming*, pages 425–443, 1994. (Cited on page 70.)

A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, 2005. ISBN 012385251X. (Cited on page 23.)

N. K. Jha. Low power system scheduling and synthesis. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 259–263. IEEE Press, 2001. ISBN 0-7803-7249-2. (Cited on page 19.)

C. Johnson. Genetic programming with fitness based on model checking. In *Proceedings of the 10th European Conference on Genetic Programming*, pages 114–124. Springer, 2007. (Cited on pages 50 and 151.)

E. De Jong and J. B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233, 2003. (Cited on pages 50, 51, and 66.)

A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, 2008. (Cited on page 24.)

J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *In Sixth Annual Workshop on Selected Areas in Cryptography*, pages 13–33. Springer, 1999. (Cited on page 135.)

R. A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. *Computer Security Applications Conference*, 0:109, 2002. ISSN 1063-9527. (Cited on page 140.)

J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976. (Cited on page 91.)

D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley, 1997. ISBN 0-201-89684-2. (Cited on page 70.)

D. E. Knuth. *The Art of computer programming, Volume 3: sorting and searching (2nd Edition)*. Addison Wesley, 1998. ISBN 0-201-89685-0. (Cited on page 7.)

P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99: Proceedings*, pages 388–397. Springer-Verlag, 1999. (Cited on page 135.)

P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113. Springer-Verlag, 1996. (Cited on page 135.)

J. R. Koza. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *IJCAI-89: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 768–774. Morgan Kaufmann, 1989. (Cited on page 5.)

J. R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 37–44. Morgan Kaufmann, 1991. (Cited on page 70.)

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. ISBN 0-262-11170-5. (Cited on pages 5, 29, 32, 33, 36, 42, 46, 50, and 87.)

J. R. Koza. Gene duplication to enable genetic programming to concurrently evolve both the. In *IJCAI-95: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 734–740. Morgan Kaufmann, 1995a. (Cited on page 47.)

J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1995b. ISBN 0262111896. (Cited on page 29.)

J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, 1999. ISBN 1-55860-543-6. (Cited on page 29.)

J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003. ISBN 1-4020-7446-8. (Cited on pages 5, 29, 51, and 52.)

J. R. Koza, L. W. Jones, M. A. Keane, M. J. Streeter, and S. H. Al-sakran. Toward automated design of industrial-strength analog circuits by means of genetic programming. In *Genetic Programming Theory and Practice II*. Kluwer Academic Publishers, 2004a. (Cited on pages 35 and 66.)

J. R. Koza, M. A. Keane, and M. J. Streeter. Evolvable hardware. In *6th NASA / DoD Workshop on Evolvable Hardware (EH 2004)*, pages 3–17. IEEE Computer Society, 2004b. (Cited on page 26.)

J. R. Koza, S. H. Al-Sakran, and L. W. Jones. Cross-domain features of runs of genetic programming used to evolve designs for analog circuits, optical lens systems, controllers, antennas, mechanical systems, and quantum computing circuits. In *EH '05: Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware*, pages 205–214. IEEE Computer Society, 2005. (Cited on pages 5, 26, and 30.)

F. Kri and M. Feeley. Genetic instruction scheduling and register allocation. In *Proceedings of the The Quantitative Evaluation of Systems Conference*, pages 76–83, 2004. (Cited on page 26.)

P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the conference on Programming language design and implementation*, pages 171–182, 2004. (Cited on page 26.)

S. Kumar, A. Jantsch, M. Millberg, J. Öberg, J.-P. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. *VLSI, IEEE Computer Society Annual Symposium on*, 2002. (Cited on page 17.)

C. Lamenca-Martinez, J. C. Hernandez-Castro, J. M. Estevez-Tapiador, and A. Ribagorda. Lamar: A new pseudorandom number generator evolved by means of genetic programming. In *Parallel Problem Solving from Nature IX*, volume 4193, pages 850–859. Springer-Verlag, 2006. (Cited on pages 70, 71, 72, 75, 78, 79, and 82.)

W. B. Langdon. Data structures and genetic programming. In *Advances in Genetic Programming 2*, pages 395–414. MIT Press, 1996a. ISBN 0-262-01158-1. (Cited on page 35.)

W. B. Langdon. Scheduling maintenance of electrical power transmission networks using genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*. Nagoya University, Japan, 1996b. (Cited on page 35.)

W. B. Langdon. Fitness causes bloat: Mutation. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 633–638. IEEE Press, 1998. (Cited on page 49.)

W. B. Langdon. Quadratic bloat in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458. Morgan Kaufmann, 2000a. (Cited on pages 5 and 48.)

W. B. Langdon. Size fair and homologous tree crossovers for tree genetic programming. *Genetic Programming and Evolvable Machines*, 1 (1-2):95–119, 2000b. (Cited on page 35.)

W. B. Langdon. How many good programs are there?: How long are they? In *Foundations of Genetic Algorithms VII*, pages 183–202. Morgan Kaufmann, 2002. (Cited on page 40.)

W. B. Langdon and P. Nordin. Seeding genetic programming populations. In *Proceedings of the European Conference on Genetic Programming*, pages 304–315, 2000. (Cited on pages 35 and 66.)

W. B. Langdon and R. Poli. Fitness causes bloat. In *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 1997. ISBN 3-540-76214-0. (Cited on page 50.)

W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. ISBN 3-540-42451-2. (Cited on pages 33, 42, 44, and 94.)

W. B. Langdon and R. Poli. On Turing complete T7 and MISC F-4 program fitness landscapes. Technical Report CSM-445, 2005. (Cited on page 40.)

W. B. Langdon, S. Gustafson, and J. R. Koza. The Genetic Programming Bibliography, 2009. URL `http://www.cs.bham.ac.uk/~wbl/biblio/`. (Cited on page 5.)

Pepijn J. De Langen and Ben H. H. Juurlink. Trade-offs between voltage scaling and processor shutdown for low-energy embedded multiprocessors. In *SAMOS*, pages 75–85. Springer, 2007. (Cited on pages 18 and 19.)

J. Y. Lee, S. I.K. Choa, and I. C. Park. Performance enhancement of embedded software based on new register allocation technique. *Microprocessors and Microsystems*, 29(4):177–187, 2005. (Cited on page 20.)

S. Leventhal, L. Yuan, N. K. Bambha, S. S. Bhattacharyya, and G. Qu. DSP address optimization using evolutionary algorithms. In *Proceedings of the workshop on Software and compilers for embedded systems*, pages 91–98, 2005. (Cited on page 25.)

X. Li, M. J. Garzaran, and D. Padua. Optimizing sorting with genetic algorithms. In *Proceedings of the international symposium on Code generation and optimization*, pages 99–110, 2005. (Cited on page 25.)

S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006. (Cited on page 95.)

S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248. Morgan Kaufmann, 1997. (Cited on page 52.)

S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213. Morgan Kaufmann, 1998. (Cited on page 52.)

Mälardalen WCET Research Group. Wcet project benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. (Cited on page 96.)

A. Marek, W. D. Smart, and M. C. Martin. Learning visual feature detectors for obstacle avoidance using genetic programming. *Computer Vision and Pattern Recognition Workshop*, 6:61, 2003. (Cited on page 35.)

R. Mathew, M. Younis, and S. M. Elsharkawy. Energy-efficient bootstrapping for wireless sensor networks. *Innovations in Systems and Software Engineering*, 1(2):205–220, 2005. (Cited on page 21.)

J. McHugh. *Handbook for the Computer Security Certification of Trusted Systems*, chapter Covert Channel Analysis. 1995. (Cited on page 135.)

P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004. (Cited on pages 91 and 96.)

Mersenne Twister PRNG. University of Hiroshima, 2009. URL `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`. (Cited on page 73.)

B. Mesman, L. Spaanenburg, H. Brinksma, E. F. Deprettere, E. Verhulst, F. Timmer, H. van Gageldonk, L. D. J. Eggermont, R. van Leuken, T. Krol, and W. Hendriksen. *Embedded Systems Roadmap – Vision on technology for the future of PROGRESS*. STW Technology Foundation, 2002. ISBN 0521474655. (Cited on page 3.)

Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000. ISBN 3540224947. (Cited on page 29.)

G. De Micheli, R. K. Gupta, and K. Gupta. Hardware/software codesign. *IEEE Micro*, 85:349–365, 1997. (Cited on page 13.)

T. Miconi. Why coevolution doesn't "work": superiority and progress in coevolution. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481, pages 49–60. Springer, 2009. (Cited on page 65.)

J. Miller and P. Thomson. Cartesian genetic programming. In *Proceedings of the 3rd European Conference on Genetic Programming*, pages 121–132. Springer-Verlag, 2000. (Cited on pages 47 and 48.)

J. Miller, M. Reformat, and H. Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006. (Cited on page 96.)

B. Mitavskiy and J. Rowe. Some results about the markov chains associated to GPs and to general EAs. *Theoretical Computer Science*, 361(1):72–110, 28 August 2006. (Cited on page 45.)

D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995. (Cited on page 46.)

Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006. ISBN 0471316490. (Cited on page 100.)

Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. (Cited on page 139.)

G. Myers. *The Art of Software Testing*. Wiley, 1979. ISBN 0471043281. (Cited on pages 91 and 122.)

J. Noble and C. Weir. *Small memory software: patterns for systems with limited memory*. Addison-Wesley, 2001. ISBN 0-201-59607-5. (Cited on page 22.)

P. Nordin. AIMGP: A Formal Description. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*. Stanford University Bookstore, 1998. (Cited on page 49.)

G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing*, 17(2):160–176, 2005. (Cited on page 151.)

M. O'Neill and C. Ryan. Automatic generation of caching algorithms. In *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134. John Wiley, 1999. (Cited on page 25.)

M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 2003. ISBN 1402074441. (Cited on pages 46, 47, and 48.)

U. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In *Foundations of Genetic Algorithms 3*, pages 73–88. Morgan Kaufmann, 1994. (Cited on pages 42 and 43.)

P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(1):3–13, 1999. (Cited on page 16.)

J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *Pervasive Computing, IEEE*, 4(1):18–27, March 2005. (Cited on page 69.)

S. Park and H. Shin. Performance evaluation of memory management configurations in linux for an os-level design space exploration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, SAMOS 2007.*, pages 24–33. Springer, 2007. (Cited on page 18.)

D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980. (Cited on page 14.)

A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996. (Cited on page 14.)

R. Poli and M. Graff. There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, pages 195–207. Springer, 2009. (Cited on page 51.)

R. Poli and W. B. Langdon. A new schema theorem for genetic programming with one-point crossover and point mutation. Technical Report CSRP-97-03, University of Birmingham, UK, 1997. (Cited on pages 42 and 44.)

R. Poli and W. B. Langdon. Genetic Programming Theory I & II. GECCO Conference Tutorial, 2009. (Cited on page 32.)

R. Poli and N. McPhee. General schema theory for genetic programming with subtree-swapping crossover: part i. *Evol. Comput.*, 11(1):53–66, 2003. (Cited on page 44.)

R. Poli and N. McPhee. Parsimony pressure made easy. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1267–1274, 2008. (Cited on pages 50 and 109.)

R. Poli, N. McPhee, and J. E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004. (Cited on page 45.)

R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. 2008. URL `http://www.gp-field-guide.org.uk`. (Cited on pages 29 and 87.)

C. Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc, 1995. (Cited on page 71.)

R. *The R Project for Statistical Computing*, 2009. URL `http://www.r-project.org/`. (Cited on page 114.)

M. Reformat, X. Chai, and J. Miller. On the possibilities of (pseudo-) software cloning from external interactions. *Soft Comput.*, 12(1):29–49, 2007. (Cited on page 87.)

J. L. Risco-Martín, D. Atienza, R. Gonzalo, and J. I. Hidalgo. Optimization of dynamic memory managers for embedded systems using grammatical evolution. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1609–1616. ACM, 2009. (Cited on page 25.)

K. Rodriguez-Vazquez, C. M. Fonseca, and P. J. Fleming. Multiobjective genetic programming: A nonlinear system identification application. In *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 207–212. Stanford Bookstore, 1997. ISBN 0-18-206995-8. (Cited on page 63.)

J. P. Rosca. Analysis of complexity drift in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294. Morgan Kaufmann, 1997. (Cited on pages 42 and 44.)

C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997. (Cited on page 92.)

C. Rusu, R. Melhem, and D. Mossé. Maximizing rewards for real-time applications with energy constraints. *Transactions on Embedded Computing Systems*, 2(4):537–559, 2003. (Cited on page 19.)

R. Sagarna and J.A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, 2006. (Cited on page 96.)

B. Sareni and L. Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary Computation*, 2(3):97–106, Sep 1998. (Cited on page 61.)

M. Sarrafzadeh, F. Dabiri, R. Jafari, T. Massey, and A. Nahapetian. Low power light-weight embedded systems. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, pages 207–212. ACM, 2006. (Cited on page 3.)

M. D. Schmidt and H. Lipson. Incorporating expert knowledge in evolutionary search: a study of seeding methods. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1091–1098. ACM, 2009. (Cited on pages 35 and 90.)

D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing, 2000. ISBN 0201737191. (Cited on page 15.)

L. Shang, L. Peh, A. Kumar, and N. K. Jha. Temperature-aware on-chip networks. *IEEE Micro*, 26(1):130–139, 2006. (Cited on page 17.)

D. Sheldon and F. Vahid. Making good points: application-specific pareto-point generation for design space exploration using statistical methods. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 123–132. ACM, 2009. (Cited on page 25.)

S. Silva and J. Almeida. Dynamic maximum tree depth. In *Genetic and Evolutionary Computation – GECCO-2003*, pages 1776–1787. Springer-Verlag, 2003. (Cited on page 50.)

A. Sinha, A. Wang, and A. P. Chandrakasan. Algorithmic transforms for efficient energy scalable computation. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 31–36. ACM Press, 2000. (Cited on page 22.)

M. Sipper and M. Tomassini. Co-evolving parallel random number generators. In *Parallel Problem Solving from Nature – PPSN IV*, pages 950–959. Springer, 1996. (Cited on page 70.)

M. Smith and L. Bull. Improving the human readability of features constructed by genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1694–1701. ACM Press, 2007. (Cited on page 51.)

S. F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980. (Cited on page 29.)

T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 781–186. IEEE Press, 1998. (Cited on page 49.)

L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002. (Cited on page 47.)

M. Stephenson, S. Amarasinghe, M. Martin, and U. M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, 2003. (Cited on page 25.)

T. K. Tan, A. Raghunathan, and N. K. Jha. Energy macromodeling of embedded operating systems. *ACM Trans. Embed. Comput. Syst.*, 4(1): 231–254, 2005. (Cited on page 151.)

R. J. Terrile, H. Aghazarian, D. Keymeulen, G. Klimeck, M. A. Kordon, and P. Allmen. Evolutionary computation technologies for the automated design of space systems. In *EH '05: Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware*, pages 131–138. IEEE Computer Society, 2005. (Cited on page 23.)

A. Thompson, P. Layzell, and R. S. Zebulum. Explorations in design space: unconventional electronics design through artificial evolution. *IEEE Transactions on Evolutionary Computation*, 3(3):167–196, 1999. (Cited on page 26.)

A. Turing. Intelligent machinery. Technical report, 1948. URL http://www.alanturing.net/intelligent_machinery/. (Cited on page 29.)

O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Coolcache: A compiler-enabled energy efficient data caching framework for embedded / multimedia processors. *Transactions on Embedded Computing Systems*, 2(3):373–392, 2003. (Cited on page 20.)

F. Vahid and T. Givargis. Highly-cited ideas in system codesign and synthesis. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 191–196. ACM, 2008. (Cited on page 13.)

A. Vargha and H. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educational and Behavioral Statistics*, 25(2):101–132, 2000. (Cited on page 117.)

K. Villela, J. Doerr, and A. Gross. Proactively managing the evolution of embedded system requirements. In *International Requirements Engineering, 2008. RE '08. 16th IEEE*, pages 13–22, 2008. (Cited on page 4.)

A.G. Voyiatzis, A.G. Fragopoulos, and D.N. Serpanos. *Embedded Systems Handbook*, chapter 17 (Design Issues in Secure Embedded Systems). Taylor and Francis, 2006. ISBN 0849328241. (Cited on page 13.)

B. Warneke, M. Last, B. Liebowitz, and K. S.J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1): 44–51, 2001. (Cited on page 69.)

A. F. Webster and S. E. Tavares. On the design of S-boxes. In *Advances in Cryptology — Crypto '85*, pages 523–534. Springer-Verlag, 1986. (Cited on pages 73 and 134.)

T. Weise and K. Geihs. Genetic programming techniques for sensor networks. In *Proceedings of 5. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, pages 21–25, 2006. (Cited on pages 62 and 66.)

C. H. Westerberg and J. Levine. Investigation of different seeding strategies in a genetic planner. In *Proc. EvoWorkshops2001: EvoCOP, EvoFlight, EvoIASP, EvoLearn, and EvoSTIM*, pages 505–514. Springer, 2001. (Cited on page 35.)

P. A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, pages 178–181. IEEE Press, 1995. (Cited on pages 42 and 44.)

P. A. Whigham and J. P. Rosca. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (TR95.2)*, pages 33–41, 1995. (Cited on page 46.)

D. R. White and S. Poulding. A rigorous evaluation of crossover and mutation in genetic programming. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, pages 220–231. Springer, 2009. (Cited on pages vii, 39, 52, and 100.)

D. R. White, J. A. Clark, J. Jacob, and S. Poulding. Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators. In *GECCO 2008: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pages 1775–1782, 2008. (Cited on pages vii, 70, and 78.)

D. Wild. Code optimisation using genetic algorithms. Master's thesis, University of York, 2002. (Cited on pages 26 and 152.)

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. (Cited on page 51.)

XScale Series, 2007. URL `http://www.intel.com/design/intelxscale/`. (Cited on page 18.)

R. Yavatkar and K. Lakshman. A CPU Scheduling Algorithm for Continuous Media Applications. In *NOSSDAV '95: Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 210–213. Springer-Verlag, 1995. (Cited on page 19.)

Y. Yoshida, B. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 265–268. ACM Press, 1997. (Cited on page 16.)

C. Zhang and F. Vahid. A Power-Configurable Bus for Embedded Systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 809–812. Piscataway, 2002. (Cited on page 18.)

F. Zhao and L. Guibas. *Wireless Sensor Networks: an Information Processing Approach*. Morgan Kaufmann, 2004. (Cited on page 21.)

S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996. (Cited on pages 22 and 149.)

E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, 2001. (Cited on pages 60, 63, and 74.)

COLOPHON

This thesis was typeset with LaTeX 2_ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

Final Version as of March 25, 2010 at 16:06.