Program Transformation of Embedded Systems

Andrew David Moss

A thesis submitted to the University of Bristol in accordance with the requirements for the Degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

50,000 words

November, 2005

Abstract

This thesis explores the use of program transformation tools in the analysis and compilation of programs for embedded systems. The intrinsic difficulty in programming for these small-scale systems arises from the discrete effects that executing an instruction produces. In larger scale systems these effects are masked by the large number of instructions executed to perform a typical action.

The central claim of this thesis is that program transformation tools can aid the programmer in understanding and controlling these effects and therefore reduce the complexity of the problem; both in difficulty and in scale.

We have investigated this method on two cases that are typical of the problems in embedded software development; software timing and precision. In the former case we have developed a novel abstract interpretation that computes the timing distance between locations in a program binary. Our new technique for analysing the structure of a program can then be combined with the set of timing distances to generate models of the timing behaviour of the program. We demonstrate this work in the context of verifying legacy applications.

In the later case we look at the problem of roundoff error in software. Given the small datasizes on typical embedded processors, the programmer is normally responsible for handling the rounding of values and ensuring that datatypes are sufficiently large to represent data without an error. We present a domain specific language that allows this property to be expressed directly. Analysis of how to propagate the variable precisions in this domain is investigated, and we conclude with a technique to automatically generate a compiler for the domain.

Contents

List of Figures

Acknowledgements

The exposition in Chapter 7 owes its clarity to John Gallagher who created a clear overview of the work, and coined the funky term *syntactic isomorphism*. All of the people who worked on ASAP have my thanks for making it such a fun project, and for coming up with all the good ideas . . .

There are many people who either helped to make this thesis happen, or made the making more fun. Thanks to Mum and Dad, you were there for me all these years, I couldn't have done it without you. I know that I said I wouldn't be a student forever, but I was nearly right. Veronica, you've made me the happiest guy in the world. I can't begin to explain how good it's been in such a few words. My life has been amazing since I met you. You mean the whole world to me, and I'm looking forward to spending the rest of my life with you. Thanks for everything babes.

All of the people in Bristol who made the department such a rewarding place to be over the years. The wearables guys; Mike, Paul, Ian and Cliff. The crypto guys, both present and those who have escaped to greener pastures; Dan, Nigel, Fre, John, Tobias, Kamel and even Pete. The Dutch Contingent, who are freakishly tall, yet interesting people; Henk, Peter, Bob and Erik. Martijn who not only offered good advice whenever I put a really bad research idea to him, but who also improved my go game immensely.

Many thanks must go to my financial backers who managed to keep me afloat by being so bad at Poker over the past few years; Alistair, Kate, Mark, Rob and Denis. Angus, who has never played poker[1] but is always good for just the one[2]

[1]This may not be strictly true

[2]This is also slightly inaccurate

drink anyway.

Doctor Granger, for being such a good mate over the past couple of years. It would have been a grey and boring place without you to liven it up. Finally, thanks must go to Henk for being such a good supervisor over the demi-decade that it took in the end. You pulled me through some of the worst times, but most importantly you taught me how to be an academic. As you put it once, but have denied saying ever since;

> Research is like looking for the edge of a cliff in a strange field, in the
> the dark. You do it very carefully, one step at a time.

Chapter 1

Introduction

Computer programs, and their study, are central to the subject of computer science. The primary concern is not usually individual programs themselves, but rather how programs can be classified into collections. The properties that are used to group programs vary widely across the subject. In one extreme the concern is the efficiency of programs; categorised by their complexity. At the opposite extreme the language community considers the commonalities in function that programs possess. These commonalities in function are reflected by commonalities in structure and form.

There are many different programming languages, and each program can be expressed in many of them. The broadest distinction between programming languages is whether or not they are universal. A universal language can encode any program. Between any pair of universal languages there must be a translation, that transforms the encoding of a program in one language into an encoding of an equivalent program in the other language.

Some languages do not strive for universality, instead they are designed with the purpose of expressing a particular set of programs clearly. The commonalities in structure and form that the set of programs exhibit are lifted into the language itself. The language becomes specialised to a particular problem domain, more suited to writing programs within that domain because the common structure is implied by the language. The common structure is removed from the program making its construction more simple.

All practical programming languages offer a level of abstraction built on top of more basic languages. A domain-specific-language (**DSL**) is more abstract than a general purpose language as it has removed the need for common structure to be expressed in programs. This abstraction is similar to the use of libraries and an application-programming-interface (**API**); common pieces of functionality are predefined so that programs are simpler to write. A DSL, rather than a library in a general purpose language, is a higher level of abstraction. The language can enforce a semantic contract on the program, for example on the ordering of operations, that a library cannot. Restricting the set of programs that can be written to those that are correct for the domain has the potential to make programmers more productive *in that domain* than in the general case. The cost that is paid for this productivity is the difficulty of translating these abstract codes into efficient executable forms.

The translation of programs from one form to another is studied in the field of Program Transformation. There are two types of problem within this field that we shall discuss in this thesis;

Analysis is a transformation from a program to a value. The value is a representation of some property in the program. Not all properties can be analysed exactly, and often we are *approximating* a program property with a value.

Compilation is a transformation from a program in one language, known as the source, into a program in a second language, known as the target. The source language is the more abstract of the two, and the target language is more executable. This transformation lowers the level of abstraction in the program making it more concrete as a step towards executing it on a real machine.

Machine code is the programming language operating at the lowest level of abstraction accessible to the programmer. A machine code program is encoded as a list of numbers. Each of the numbers that can be stored in the list has a defined function. These functions manipulate the state stored within a processor. There are no data abstractions in the language — all types of data are explicitly encoded as numbers and manipulated using arithmetic and logical operators. Programming at this low level is verbose and prone to error. All operations on data must be manually encoded as operations on fixed size bit vectors. This manual encoding is a repetitive task, and all locations in the program manipulating a common value must agree upon a common encoding and decoding.

All of the effects that the program produces are precisely defined in the low-level executable form. This precise definition may be specific to a particular model of processor that implements the machine code. Each of the listed numbers in the program represents an instruction for the processor. The instructions each have well defined effects on particular models of processor — the target architecture. Given a particular machine code program and a specification of the target architecture it is possible to decide exactly what effects the execution of the program will produce by execution.

The main effect of a program is its explicitly defined function. Each program is a function, or more generally a sequence of functions. Each of the functions is a mapping between two states. In the classical view of an algorithm the state is initialised to contain some input data, and on termination of the algorithm the state contains some output data. The function that the program defines is this mapping from input (state) to output (state). More generally a computer program may accept new input during its execution, and produce output while executing. In this case the program still defines a mapping between states, but some states contain information that is exchanged with the environment.

The syntactic constructs within a language allow the programmer to construct this functional mapping. Depending on the language style these constructs may be the direct composition of functions, or definitions of logical mappings, or even imperative state updates. In every case the constructs are well defined and each combination of them has a defined effect upon the state mapping. Regardless of the language style the programmer can explicitly see how their modifications alter this explicit effect of the program.

The execution of a program has effects other than this explicit function on state. These effects are measurable properties of the execution of the program that are not directly specified by the programmer. We will use the term *side-effects*, which is common in some fields.

The side-effects under consideration are all deterministic when a low-level program form is executed on a particular target architecture. Normally these side effects are not defined in higher level languages. This means that the programmer cannot reason about the code that is written in the high-level language and the effect of the program when it is executed as a low-level form. The problem domains that we investigate particularly highlight this type of side-effect, as the correct execution of the program depends upon these effects.

We investigate these side-effects in the two previously described problems in program transformation; analysis and compilation. Our concern in program analysis is determining which side-effects a program defines. The programs that we consider are low-level executable forms — otherwise we must consider the specific compilation that will turn the program into a form that defines the side-effects. Part I of this thesis defines the analysis, and applies it to programs that are specific to a single model of micro-controller.

In program compilation we investigate making an effect that is normally implicit, in a language, explicit for the programmer to manipulate. To ensure that the high-level program is compiled into a form that correctly reflects the side-effect it is necessary to perform program analysis on the high-level form. This analysis ensures that the side-effect can be correctly represented in the low-level program. The compilation process then preserves the property whilst mapping from the high-level program representation to the low-level. Part II of this thesis investigates the design of such a language, the necessary analysis, and the generation of executable code.

1.1 Problem Domain

The programs that we consider are motivated by a specific problem domain; wearable computing. A wearable computer is worn by the user and thus disconnected from mains power. Generally such systems are battery powered. The aim of such computers is to use information about the user's current activity to aid the user by performing useful tasks. The information about the user's activity is collected by a set of sensors that monitor the context of the user. This context contains the user's position, orientation and movement.

Each program monitors the user's context and detects when the user is performing specific tasks. At these times it presents information that can aid the user in their activity. The communication with the user can be explicit, presenting audio or visual data, or implicit, such as the monitoring of activity. The aim of wearable computing is to explore how computing that is pervasive to the user's environment can provide a useful aid to the user.

One example of a situation in which a wearable computer can aide a user is described by Edward Thorp [75]. In a joint project with Claude Shannon they designed and built a wearable computer that could predict the octant of a roulette wheel that the ball would land in. In this example the communication between wearer and computer is explicit, passing timing information to the computer and receiving octant estimates in return,

The wearable computer [61, 65, 60] that motivates our research is the Bristol Cyberjacket. The overriding concern in this system is energy conservation. The battery that powers the system is a limited resource, and once it is depleted the system cannot function until the battery is replaced or recharged. For the Cyberjacket to be a seamless part of the user's environment these recharges need to be as infrequent as possible.

The system is made of a single main computer and a network of simple devices, as shown in Figure 1.1. Each device contains a sensor or actuator, and some local processing capability in the form of a micro-controller. The main computer is several orders of magnitude more costly in power-consumption than the simple devices. The simple devices each form an interface between the main computer and the user's environment. Each of the sensor devices provide input for processing on the main computer. The role of the micro-controller on each device is to filter the input as much as possible so that the computation the main computer has

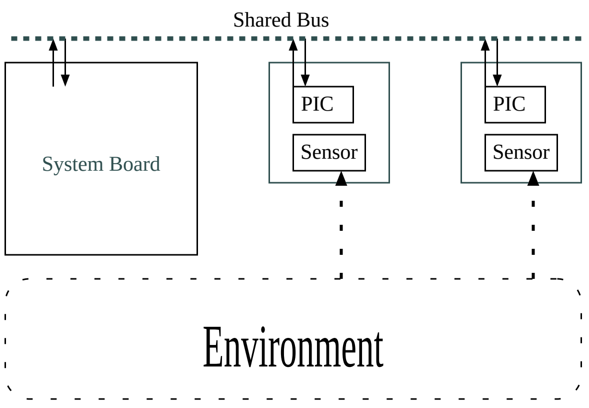


Figure 1.1: Sensor architecture

to perform is minimised. This filtering allows the main computer to suspend its operations whenever possible and reduce its power-consumption.

The programming model on the main computer is *event triggered*, the input that is transmitted from the sensor devices form events that are processed by the computer. Filtering the input into a more abstract form means that each sample of the environment does not require processing by the computer. Only the input samples that relate to interesting changes in the user's context form events for processing, whilst the remained are filtered out of the stream of events.

In contrast, the programming model on the sensor devices is *time triggered*. Each of the physical sensors needs to be polled at a real-time interval. The micro-controller operates continually, matching the speed at which it processes its program to the rate at which the sensor requires polling. When events are communicated to the computer a bus is used that connects all of the devices together. The protocol for communicating over the bus is also time-based and the micro-controller must access the bus at predefined times.

Each of the programs that is written for the sensor devices performs three functions. Instructions on the micro-controller provide external access to hardware devices. These instructions are used to interact with the sensor. The interactions occur at periodic points in time. The programmer must ensure that each sequence of instructions in the program takes the correct length of time to execute, in-between these interactions. Failure to interact with the sensor at the correct time would corrupt the input.

Another piece of hardware that is attached to the micro-controller is a common bus between the devices in the system. The sensor device can read the state of the bus to receive messages from other devices in the system, and can alter the state of the bus to transmit messages. The protocol for accessing the bus is a time-based protocol — there are no explicit acknowledgements that messages have been received. Instead timing guarantees ensure delivery. Each device holds the transmission state for a sufficiently long time that the receiver will receive the message. For these timing guarantees to hold we must be able to reason about the length of time that parts of a program take to execute and how frequently actions such as checking the state of the bus occur.

While the sensor is performing its two interactive functions it must also execute a filter program. Filters are programs with simple control flow that manipulate mainly numerical data. They can be used to smooth a noisy signal for further processing, and to identify properties of the data that are useful to the system. The filters that are considered on the Cyberjacket are well-known; Kalman filters are used to model the attributes being sensed and neural networks are used to recognise patterns in the data.

Both of these filters have suitable control flow and numerical operations to be applicable to the analyses and transformations that we study in this thesis. The Kalman Filter has been used as the motivating example throughout the thesis, but the techniques apply equally well to other filters, and to neural networks in particular.

Our aim in studying these applications is to increase the level of automation in designing the types of system that we have described. The current state-of-the-art for building these systems involves programming directly in a low-level assembly language [61, 65, 60]. While the subject has advanced beyond this state in other types of systems there are still specific aspects of these applications that are difficult to express in a high-level language.

In particular their need to communicate with their environment at well-defined times is difficult to achieve with current languages. Plenty of work exists on compiling real-time systems when the granularity of time is measured in 10s of thousands of instructions. In our domain the timing intervals are measured in dozens of instructions, and the programmer must have a fine-grained control over the scheduling and placement of instructions to meet timing constraints.

Limiting the programmer to a low-level language so that they may express fine-grained timing constraints is counter-productive. In particular the low level of abstraction makes it difficult to write correct numerical processing code. Mainstream languages, such as C, make it difficult to write these filters effectively. The language makes no guarantees on the rounding modes of the underlying hardware or the sizes of the representations that are used. We seek to remedy these failures by providing explicit guarantees to the programmer for the size and rounding of the numerical representations that their program manipulates. At the same time we seek to raise the level of abstraction beyond that in a general purpose language by tailoring a DSL to this specific problem of writing filter code.

Reaching these aims will increase the ability of system designers to write these 'difficult' low-level processes. By making explicit the properties of the low-level language that they depend upon for their programs to function correctly we will make it possible to design these processes at a higher level of abstraction — without losing the ability to ensure their correctness.

1.2 Target Architecture

The parts of the Cyberjacket system that motivate our work are the sensor boards. The most resource constrained part of the system is the micro-controller used on these boards. While it does not possess a large amount of storage, and is not capable of high performance, it possesses sufficient resources to allow programs that are an interesting target for analysis and compilation.

The micro-controller that we consider is the PIC-16F84 [47]. This micro-controller is of interest because it offers a good ratio of performance to power consumption. For this reason it is a common choice of controller within the wearable and robotics communities, and hence, using the PIC as a target validates that our techniques are applicable to real-world problems.

Low power-consumption is achieved in the PIC by a simple design. The PIC is an example of the simplest register based processor design; there is a single working register called the accumulator which forms one of the operands in every instruction. This property of the PIC design makes efficient code generation difficult. In comparison a RISC design allows every register to be used as a source of operands, or as a destination to hold results. More instructions in the program are devoted to shuffling data between registers so that one of the source operands is always in the accumulator. This reduces the code density of programs, and as a result they are longer than equivalent programs for other architectures.

The PIC ISA contains 35 instructions, which provide data movement, bit manipulation, logical testing and control-flow manipulation. The instruction set does not contain the usual predicated jump instructions. The only jumps used are unconditional, instead there are bit-testing instructions which will test the status of a bit in a register and conditionally skip the next instruction on the state of the bit. These skip instructions can be combined with jumps to construct the normal predicated branches by testing the processor flags which are contained within a control register. The need to create conditional jumps from a combination of instructions in this way is another factor that reduces the code density of PIC programs.

The design of the storage on the PIC is an instance of a Harvard Architecture; the data storage and program storage are divided into two separate buses. Both stores are small by modern processor standards. The data storage is 127 8-bit registers, of which half a dozen are reserved for special functions. This leaves the programmer around 1000 *bits* of data storage. The instruction set is encoded into 14-bit words, and in the program store there is space for slightly over 1000 instructions. These limits on the data size and program size of programs are both hard limits — a program that exceeds them cannot execute on a PIC.

The reward for such tight constraints on the instruction set of the PIC result is a power consumption of just 2 mW per million instructions. The low power consumption is a direct consequence of the simple logic required to implement the instruction set. For comparison the main computer used in the Cyberjacket has a processor that consumes 2W during operation — 1000 times more power. The processor in the main computer is itself many times more power efficient than a desktop machine, in which a power consumption of 100W is not uncommon for a processor.

1.3 Organisation of Thesis

The material in this thesis is organised by which side-effect is under study. There are two parts, the first being comprised of Chapters 3 - 4, and the second being Chapters 5 - 7. Each part focuses on a single side-effect that is required to correctly implement the system in our problem domain. In both parts the implementation is in the machine code of the target architecture. Each of the parts shows how to apply program transformation techniques that allow the designer to explicitly reason about the side-effect. The goal in both cases is to automate more of the design and to allow the designer to work at a higher level of abstraction.

Part I : Timing Semantics The first effect that we consider is timing. Each program that is executed on the target architecture takes some amount of time. The length of time that the execution takes is implied by the program, but it is not explicitly manipulated in the source code. Higher-level languages do not allow the programmer to explicitly manipulate the length of time that execution of their program will take. There are usually library routines, e.g. a `sleep` function or timers, that allow the program to be suspended for an explicit length of time. Languages in the style of Esteral allow explicit time lengths to be assigned to scopes in the program. This solution works well for coarse-grained control of timing, but is not suitable for fine-grained control, in which the timing constraints may partially span several scopes.

In a typical timing problem the program must perform some function that entails executing instructions. These instructions take some length of time. The effect that these instructions produce, i.e. interaction with external devices, is separated by periods of time that are large in comparison with the computation time. The length of the computation is still hidden from the programmer but because it is so small in comparison with the periods of operation it can be ignored. In the timing problems that we consider the separation times are of the same order as the computation times. When the problem becomes this finely grained the computation times are no longer negligible and must be accounted for by the programmer for correct operation.

Part I focuses on the problem of analysing the timing behaviour of existing program code. By analysing the machine code directly, and with respect to a specific target architecture the timing characteristics of the code can be made explicit. The benefit of analysing machine code is that the output of any compiler can be used. Although this technique does not allow the programmer to reason about the timing effects directly in their source language, it can be applied to both new code and to legacy code. In the case of legacy code the benefit is verification that the code operates correctly.

Reasoning about the timing behaviour of programs is hard. It is not possible to provide an exact solution for all programs. If it were possible to do so then the halting problem could be decided for all programs — simply check if the length of execution time is finite. This presents a problem as we cannot provide an analysis that gives accurate answers on general problems. Instead we study an approximation of the timing problem that gives precise results on the set of filter programs that we are concerned with.

Part I is not directly concerned with the problem of compilation from a language with explicit timing behaviour. The analyses detailed would however support a compiler that performs this task. Determining the length of time that individual code fragments take to execute on the target architecture is an important intermediate step towards generating code with exact timing behaviours.

Part II : Precision Semantics The second effect that we consider is numerical precision. Computers use a finite representation of numerical values. Whenever a pair of these values are operated upon, the result produced must be stored as another finite representation. Restricting the domain of the values to this closed form produces an error in some cases. When the correctness of the program depends on accuracy in the values being computed, these small errors can accumulate and destroy that accuracy.

Most high-level languages offer two representations of numbers. Either integers stored in the size of the machine word, or an IEEE floating point representation. In these two representations the accuracy of the number being represented is invisible to the programmer who cannot explicitly manipulate the accuracy of the expressions being computed.

Although there are some high-level languages that avoid this problem by computing on an arbitrary-sized representation there is a performance penalty for abandoning fixed sized representations. We present a DSL tailored to the simple numerical computations in the problem domain. The programmer manipulates values of known precision, and the language guarantees that precision is preserved to the amount specified by the programmer.

Program analysis is performed to determine the sizes required for each representation in the program. This analysis uses a specification of the precision at the interfaces to the program; the input from, and output to the environment. From these annotations the analysis automatically infers the necessary amount of precision that is required for the rest of the program to guarantee a correct result.

Compilation using these fixed-size representations preserves the performance of the program but presents a challenge. Rather than a small set of known representations in the language, each representation can be parameterised with a level of precision. The code that is generated by the compiler for each operation is therefore sensitive to the context that operation occurs in. This challenge increases the complexity of the compiler implementation. Our solution is a method of compilation using generative programming that automatically constructs a compiler for the specific set of operations in the program.

The result is a high-level language tailored to our specific problem domain that compiles into efficient low-level code. This language and compiler automate the design process in the problem domain.

1.4 Contributions

This thesis explores the use of program transformation tools in the analysis and compilation of programs for embedded systems. The central claim of this thesis is that these tools reduce the complexity of the problem; both in difficulty and in scale.

Applying this method to the cases that we have studied has resulted in several contributions that either demonstrate the reduction of complexity in the compilation problem directly, or form auxiliary analyses that support a solution to the compilation problem.

- We construct an analysis of machine code programs that computes the execution time(s) between two points. This low-level analysis uses a description of the timing semantics of the target instruction set to compute the set of times that execution takes along all paths between the two points. The analysis is presented within the Abstract Interpretation framework of program analysis. We have applied the analysis to legacy code in the problem domain and show its use in verifying the timing behaviour of programs.

- We define a loop detection algorithm that applies to the control flow graphs of programs. Loops in the control flow are found and organised into a nesting hierarchy that reflects the structure of the program. This transformation from compiled program to source structure has applications in decompilation and reverse engineering. We show that the nesting forests computed by the algorithm are more precise than previous work in the field, and that the algorithm can be applied to a wider range of control flow graphs than previous work.

- We demonstrate a method for analysing the precision of values in filter programs using a program transformation technique to map the DSL program into a Constraint Logic Program over a Finite Domain — CLP(FD). The CLP(FD) program is executed by a CLP solver to produce the necessary precision constraints for each program variable. The analysis is more precise than previous data-flow approaches in some cases.

- We show the automatic generation of a compiler for a domain specific language using two stages of specialisation and a carefully crafted interpreter. This technique shows that the specialiser can perform a sufficiently precise analysis of the interpreter that no explicit staging is required to produce a result. The technique is used with program transformation tools written in a separate meta-language, and then applied between the precision DSL and the language of the target architecture. The specialiser used is an off-the-shelf product that requires no modification to map between the two languages.

Chapter 2

Background

A program defines a series of operations that can be performed in a mechanical manner. Each of the operations is well defined to allow this mechanical implementation — no intelligence is required to interpret what effect the instruction should have. Each program is encoded as a structured piece of data. The *semantics* of a program are defined in many ways, but the common aspect of these definitions is an association between the meaning of the program and the effect produced from executing the program. The semantics of each program is dependent upon the semantics that we attach to each operation, and the semantics of the structure that encodes the operations.

The operation definitions and structural semantics of a particular encoding is called a *programming language*. Languages are grouped into several broad types according to the structure used. The most basic languages use a list of instructions, the ordering of the list being the structure of the program. More commonly used languages structure a program as a tree or a graph.

Programs that are represented as trees are termed structured programs. Confusingly, programs that are structured as graphs are conventionally known as unstructured programs. This terminology is used because there are fewer restrictions on graph-structured programs and so they can take more possible forms.

A defining characteristic of a programming language is the level of abstraction at which programs are represented. Real computer implementations, in comparison to theoretical models or languages, perform very simple operations — but they perform a great deal of them at high speed. Languages that are directly executable on machines are tied to these simple operations. These languages are the low-level languages. Higher-level languages are designed around their use by humans, rather than for direct use on a machine. These higher-level languages perform more complex operations, but each operation requires many of the simple operations of a low-level language to perform on a machine. The repetitive encoding of these sequences of low-level instructions is hidden from the programmer by the abstractions in the high-level language.

Program Transformation is concerned with programs that operate upon, and manipulate these structured program forms. There are many reasons why we wish to manipulate programs; one of the main goals is the translation of a program from one language to an equivalent program in another language. When the source language is of a higher-level than the destination language, this transformation is called *compilation*. This transformation allows a programmer to work in a higher-level language, and have the machine automatically translate their program into an executable low-level form. The aim of Program Transformation is the discovery and implementation of general algorithms; transformations that can be applied to programs written in a wide range of languages.

2.1 Static Analysis

Analysis is a program transformation that takes a program as input and computes the value of a property of the program. Whilst computing some extrinsic property of the program, such as the size of the program, would constitute a trivial analysis, this is not a useful program transformation. Instead Program Analysis is concerned with the intrinsic properties of programs. These intrinsic properties of the program are defined by its semantics rather than its syntax. Program Analysis studies the algorithms that compute these intrinsic properties of the execution of the program, and make them explicit as computed values.

The simplest type of analysis is one that computes a single boolean value that expresses a property of a program. Such an analysis *decides* a property of a program. The binary properties that can be computed by an algorithm are decidable properties. Not all of the properties that we would like to compute for programs fall into this category.

The first program analysis to be defined and reasoned about, computed the decision problem of halting. The *halting problem* is to decide whether a program executes within a finite length of time and terminates, or if it continues its execution forever. Turing [76] showed that the halting problem is undecidable on general programs. While it may be possible to write a decider that uses program heuristics and other knowledge to decide the halting problem for some programs, there is no decider that can accurately analyse *all* programs.

Rice's Theorem [64] is an important generalisation of the halting problem. The Theorem is proved on recursive functions, rather than Turing Machines, and applies to the functions that algorithms compute. Rice proved that all non-trivial properties of partial recursive functions are undecidable. A trivial property is one that is uniformly true, or false, of *all* programs in a language. So non-trivial properties are all those than can be true or false for different programs within the language. This result presents a challenge for static analysis as all of the properties that we seek to analyse are non-trivial, and therefore undecidable in the general case — over all possible functions.

Applying a static analysis to programs containing recursion requires a method of finding invariant properties that executes in a finite length of time. The methods that we consider are all lattice-theoretic.

A lattice $(L, \leq)$ is a partial order defined over a set of values so that every pair of values has an infinium $\forall x, y \in L : (x \wedge y) \in L$, and a supremum $\forall x, y \in L : (x \vee y) \in L$ within the set. The infinium is the greatest-lower bound between the two values. This is the largest value, with respect to the partial order, that is $\leq$ both values. The supremum is the least-upper bound between pairs of values; the smallest values, with respect to the partial order, that is $\geq$ both values.

Lattices are useful theoretical constructs, but in order to write tractable analyses we must limit ourselves to complete lattices. A complete lattice contains the infiniums and supremums of all value pairs — the least-upper bound, and greatest-lower bound operations are closed on the set within the lattice. So every subset of the complete lattice has a least-upper bound, and a greatest-lower bound within the set of values. In practice this often involves adding values to the set that is being computed over, conventionally these values are called $\top$ (top), and $\bot$ (bottom) and their purpose is to ensure the lattice is of finite height.

Least-upper bounds are used as safe approximations of properties. If we cannot establish the exact value of a property, perhaps because it is undecidable, then we can use least-upper bound of the set of possible values as an approximation that we know is definitely as large as the property we seek.

The standard method for finding least-upper bounds of program properties is to compute the analysis over a set of current values until it reaches a fixed point. Fixed points are more easily described as properties of a function. Consider the function f:

$$\exists x \ : \ f(x) = x \tag{2.1}$$

In trying to approximate the function f we may feed in various values of x, and then use the resulting value $f(x)$ as x in our next estimate. For certain types of functions we are guarenteed to reach an x for which $f(x) = x$. At this point we have found a stable input to the function which is a fixed point of the function.

When we are analysing programs the analyses that we perform are not normally functions. In particular supplying a particular program state to the analysis can result in multiple possible states as output. These multi-functions are evaluated over a current set of program states, resulting in a new set of program states. For certain types of multi-function there is a fixed point analogous to the functional case, in which the set of output states is equal to the set of input states. These stable sets of program states are approximatations to the semmantic function that we are analysing, specifically these sets are the *least fixed-points* of the analysis as they are being approached from beneath. More importantly these approximations are *conservative*, they contain a superset of the possible program states that contains all possible valid states.

Within this thesis we use the term *fixpoint*, rather than the more standard term fixed-point. This is purely to avoid confusion in Part II when we look at fixed-point arithmetic.

This connection between the semantics of programs and the least fixpoints of functions has been formalised by Cousot et al. as Abstract Interpretation [14]. The program property under analysis exists in the *concrete domain*. The values of this property may be, and usually are, an infinite set. Hence an approximate analysis is required that can terminate in a finite length of time.

A pair of functions is defined to map between program states in the concrete domain and an abstraction of the program states. The abstraction folds separate concrete states together into a single abstract state, losing information. The abstract domain is finite in size, although some analyses may use a sequence of abstractions with intermediate abstract domains that are still of infinite size.

A partial ordering is defined over the concrete program states, according to the property values in each state. The abstract domain is formed so that the lattice of abstract property values is complete. If the pair of maps, between domains, are order preserving then performing analysis on the abstract domain will produce a result that is a safe approximation of the property in the concrete domain. The abstract domain is guaranteed to contain fixpoints by the Knaster-Tarski theorem [73] as the lattice is of finite height, and the analysis is chosen to produce a monotone function over this lattice — new states will be added to the analysis, but not removed.

In the body of this thesis we consider two program properties that we wish to analyse statically. We will use both the Abstraction Interpretation method, and also the more direct approach of constructing a data-flow analysis. Abstract Interpretation is the formalisation of this data-flow approach.

2.2 Timing Analysis

The problem of constructing a model of the execution times of a program's statements is generally undecidable. In order to form a decidable problem it is necessary to make assumptions about the type of program being analysed, and to place appropriate restrictions on what the program can do within the context of these assumptions. The set of assumptions that are made will define not just the restrictions on what the program can do, but also the form that the result of analysis will take.

The study of algorithms uses a transformational model of computation, such as a Turing Machine. The assumption is that input is provided to an algorithm prior to execution, which then proceeds for some finite length of time before terminating and producing output. Algorithms that continue to execute forever are considered to be divergent, and to represent an error. This model is a universal description of what can be computed by functions and algorithms.

Not all programs in the real world meet this theoretical model. In particular the class of *reactive* programs conduct an ongoing dialogue with their environment. Input and output of data is interleaved with computation. The computations between the interactions can still be modelled by a Turing Machine, but the behaviour of the program including its ongoing interaction cannot be represented.

If we assume that the program we are analysing is transformational, rather than reactive, then we will assume that correct programs terminate in order to produce a result. In this case we are less interested in the structure of the timing model, and more interested in analysing whether the extreme bounds of the model fit within a set of constraints. The more common case is checking that the upper-bound of execution time is within a constraint, an analysis called Worst Case Execution Time (**WCET**).

Reasoning about the timing bounds of code sequences in high-level languages resulted in Shaw's method [68]. The method introduced the idea that upper and lower bounds on the execution time of programs can be derived through an automatic analysis. The bounds on individual statements in the program are combined according to the semantics of the language to produce bounds on the overall program. These timing schemas extend Hoare logic, a set of conditions that are asserted to be true before and after execution of each statement, with clock variables to assert program invariants involving time. Shaw's method was applied to reactive programs that communicated, and synchronised with, their environments.

Later work [51] provided experimental evidence that the technique produced tight bounds on realistic programs.

The MARS group investigated building a distributed system with deterministic, guaranteed timing behaviour. The project integrated the steps from programming individual tasks with dependable timing behaviour to the operating system required to schedule them. A tool [56] calculated the maximum execution time of each program from source code. Puschner used explicit bounds to solve the problem of determining the number of loop iterations. The concept of *markers* is introduced, which are assertions on the number of times that control-flow may pass a point within a scope. The expressions defined constants statically, and code is inserted by the compiler to check the condition holds them at run-time. Markers increase the precision of the maximum bound by eliminating some control paths that are possible within the bounds on the loops iterations.

The initial problem considered in the WCET field was how to restrict the possible control-flows through a program in order to compute an upper bound on timing. Early approaches [36, 48, 56, 68] all use static loop bounds on loops, and annotations to the program which allow the analyser to disregard sets of possible executions. The programs that can be written in this way — with static data-structures and bounded recursion — are interesting because this is how embedded software must be written.

WCET has matured as a field [57] in the decade and a half since the earlier work. The model of program execution has been refined to consider the effect of architectural features; pipelines [29], caches [78, 38, 39, 80], superscalar despatch [40] branch prediction [13] and RISC designs [41]. The source languages being analysed have become more complex with automatic annotation of statement bounds in a wide range of cases. The mapping from the source languages onto the execution model has become more precise through the elimination of infeasible code paths and investigation of the mapping from source annotations onto the underlying machine code.

A WCET analysis is typically split into three phases, reflecting the areas of progress made in the field. The names of the three phases vary between researchers, but the partitioning of the analysis into these phases is generally the same; Low-Level Analysis, Flow Analysis and Calculation.

2.2.1 Low-Level Analysis

Cycle accurate timing information is extracted from a low-level representation of the program. Usually the representation is either machine code for the specific architecture, or an intermediate form within a compiler. The Low-Level Analysis models the architectural features of the target machine to improve the precision of the bound computed.

Pipelining increases the rate at which a processor can dispatch instructions. The time for each instruction to execute remains the same but the number of instructions that can be processed in a given time increases i.e. the latency remains constant but the throughput is increased. The bounds computed by WCET analysis will be very pessimistic if the overlap between instructions in the pipeline is ignored.

Lim et al. [41] adapt Shaw's Timing Schemas to apply to a RISC design. Specifically they introduce worst-case timing abstractions (WCTAs) which model the reservation of the units within the processor at the beginning and end of basic blocks which are emitted from compilation. WCET analysis proceeds on these blocks, rather than original source-level statements. Later the work was extended to cover multi-issue processors [40].

The global interactions in a timing model are investigated by Lundqvist [43], who found that timing anomalies can occur; a shorter execution time for some instructions causes a longer overall execution time. For example a cache hit, instead of a cache miss, can increase the overall execution time. Although the local execution around the cache hit is accelerated, the change in state of the cache can slow down other regions of the program. Experiments show that access to data-structures in many real programs is predictable, and that WCET bounds can be improved.

2.2.2 Flow Analysis

Flow Analysis is concerned with collecting flow information for the program. Each flow fact defines an infeasible execution path in the program. An infeasible path is a valid path in the control flow of the program, but which cannot be taken at runtime because of logical dependencies between states in the program. Examples are dependencies between variables that control different conditional paths, such as if-then-else structures, or loop bounds. Discovery of infeasible paths allows their contribution to be removed from the WCET to produce a more accurate timing bound for the program. Consider:

```
if(x>100)
    f();
if(x<100)
    g();
```

If we assume that `f()` does not affect `x`, then the two function calls are mutually exclusive. The static control-flow of the `if` constructs allows a path through both calls, but there are no values of `x` that will produce such a path at runtime. This is an example of an infeasible path.

Flow information derived from the program must be *safe*; no feasible paths through the program should be marked infeasible. Within this constraint it is desirable for the Flow Analysis to be *tight*; attempting to minimize the number of infeasible paths reported as feasible. In static analysis in general these properties are called conservativeness and precision, respectively.

Automatic extraction of flow information increases in difficulty as we consider languages with more general control-flow. For this reason manual annotation is used to complement automatic methods. Automatic methods have to be conservative, which can make them less precise than is desirable. Allowing manual annotations to override automatically extracted facts is a method of increasing the tightness of the result, and leaves the guarantee of safety to the user.

Extraction of flow information uses an abstraction of the states during program execution. The dynamic runtime data for the program is unknown and so must be abstracted into a form that can be reasoned about. Several different abstraction techniques have been applied to the problem; partial evaluation [49], abstract interpretation [26, 29, 80, 5, 19, 74], symbolic execution [11, 6, 42, 13, 7] and constraint solving [38, 39, 50, 74].

2.2.3 Calculation

Calculation combines the low-level timing information with the flow-facts derived from the program. These two sets of information are used to produce an estimate of the worst case execution time. Broadly, calculation can be split into three types. *Tree-based* methods assign timing bounds to syntactic elements in the program parse tree, such as Shaw's [68] timing schema. *Path-based* methods explicitly represent the possible paths through a program and search for the longest. *Implicit Path Enumeration Technique* (**IPET**) uses an algebraic representation of the program execution, each block i is given an execution time t_i, and a counter c_i of the number of activations. The sum $\sum_i t_i \cdot c_i$ is maximised using constraints on the program structure that reflect the restrictions of control-flow between blocks, to give the longest execution time.

A parametric calculation of WCET produces a formula over variables unknown at analysis time. This dynamic method yields a result as an equation of unknown parameters; possibly results of data input at run-time, control-flow that cannot be determined statically, or details of low-level hardware timing such as cache hit ratios. Presenting the result of the analysis as a formula over the relevant variables may provide insight into the effects of those variables at run-time.

The CiaoPP system [32] is a pre-processor for Prolog that features extensive verification tools. Using Partial Deduction and Abstract Interpretation, the system can verify many compile-time and run-time properties of programs. Of particular relevance are the complexity analyses that can produce both upper and lower bounds for logic programs. These results are parameterised by expressions over the input data that the program operates on at run-time.

Vivancos [78] relaxes the restrictions on known loop-bounds in programs to produce residual formulas from a static timing analysis. The formulas are evaluated at run-time to make scheduling decisions. The substitution of dynamic data into these static formulas produces a more precise timing analysis than a static method alone, without the overhead of a purely dynamic method. Engelen [77] introduced a framework for applying parametric analysis to a wider range of loop types.

2.2.4 Comparison of WCET and our Timing Analysis

The field of WCET that we have briefly surveyed is the closest work to the Timing Analysis presented in Part I of this thesis. The work that we present differs significantly from WCET, despite tackling a similar problem. While WCET is concerned with determining the tightest possible estimate of the upper bound on execution, our work is concerned with the structure of the set of execution times. A WCET analysis must be performed on transformational programs — a program must terminate in order to have an upper bound on execution. Our work analyses reactive [54, 53] programs, those that are designed to continue an ongoing dialogue with their environment rather than terminate.

Despite these differences there are several similarities and connections between the work and WCET. The division of the Timing Analysis into three phases mirrors the organisation of WCET. The Low-Level Analysis performs a similar function to the Analysis of Target Code described in Chapter 3. The Analysis and Detection of Loops in control flow graphs that we cover in Chapter 4 removes infeasible execution paths from the set of timing results, this is analogous to Flow Analysis.

Finally, the equation generation that is used to construct the generators of the timing set fulfills a similar role to parameteric Calculation in WCET. In particular it is similar to the complexity analysis in CiaoPP. While the Timing Analysis is concerned with the computation of the low-level execution costs, CiaoPP produces parametric equations for the high-level control flow. Although both tools solve different problems there is a large amount of common technqiue between them — and potential for integration.

2.3 Precision Analysis

Variables in programs commonly hold representations of numbers. There are two properties of the representation that determine which set of numbers can be used, and how closely the operations on the representations match the underlying operations on the numbers. The dynamic range of a variable is the difference between the largest and smallest number that can be represented. The granularity of a variable is the smallest difference between a pair of numbers that can be represented.

Fixed point variables only store the mantissa, the exponent is constant. As there are a constant number of fractional bits, the granularity is uniform across the dynamic range. Floating point variables store both a mantissa and an exponent. The granularity is more coarse for values of greater magnitude, and finer for smaller numbers.

Both types of representation are commonly used in workstations. A floating-point representation allows the programmer to ignore the relative magnitudes of variables within their program. It is not necessary to manually scale the representations of variables into the same range to operate upon them. The hardware implementation performs this chore. Moving this functionality from software into hardware has the advantage that code can be written more clearly.

One disadvantage arises directly from the scaling operations being invisible to the programmer. As rounding and scaling are dependent on the particular value representations and are hidden from the programmer, the arithmetic operations lose their associative and commutative properties. While the programmer has gained clarity on one front, it has been sacrificed on another.

Another disadvantage of a floating-point representation is that the implementation is more complex. There is more work in executing each operation, as additional shifting and normalisation has to be performed. It is possible to mitigate this cost in performance by dedicating silicon area to the implementation and increasing the clock-speed. This approach cannot be used in an environment that must restrict power usage. One common application for numerical processing is in micro-controllers and DSPs which are constrained by small limits on implementation sizes and energy consumption [61, 65, 60].

Precision Analysis investigates how the advantages of both representations can be combined. It is desirable to remove the chore of manually scaling values from the programmer, and to bound the sizes of representations. If this can be achieved statically then the programmer can manipulate values expressively in their program, which can be compiled into an efficient form using only integer operations.

There are two fields in which Precision Analysis has been studied. In Embedded Systems the goal is to provide an accurate translation from a floating-point form to a fixed-point form — possibly reducing the number of words necessary to store each value. In Hardware Synthesis the concern is to remove as much redundant computation and storage as possible. Each bit that is removed will result in a smaller resultant circuit as each operation is being generated for the specific source and target operands.

2.3.1 Embedded Systems

In a translation from floating-point to fixed-point format, the position of the binary point must be fixed for each variable. The number of bits above the point is referred to as the *integer bitwidth*, the bits below the binary point are the *fractional bitwidth*. The previous work in the literature assumes that both the integer bitwidth and the fractional bitwidth are positive, implying that both parts of the representation are contiguous with the binary point. In general this need not be the case. Increasing the integer bitwidth of a variable increases the dynamic range that it can represent. As the number of bits in a variable is fixed by the word size in the architecture, an increase in the integer bits produces a decrease in the fractional bits. The dynamic range is increased at the cost of increased quantisation error in the variable.

There are two approaches to the problem commonly used within the literature. The *analysis* approach propagates bounds between variables using the operations in the program. The bounds produced for each variable are conservative, which guarantees that no overflows can happen during the execution of the program. The *simulation* approach uses a statistical model of each variable, computed from the distribution of the input values. The statistical bounds are not conservative, and depend upon the input values supplied during simulation. There is no guarantee that the program will not overflow during execution but the bounds produced are more aggressive, allowing a more efficient execution.

Analysis Approach FRIDGE [81, 35], (Fixed-point pRogrammIng DesiGn Environment) is a source-to-source transformation. The input programs are written in ANSI-C using floating-point arithmetic. An automatic tool converts the input programs into Fixed-C, a superset of ANSI-C with a new data-type that represents fixed-point numbers. Internally, interval arithmetic is propagated over the program variables. The Fixed-C programs are transformed back into ANSI-C, introducing the necessary shift operations to align values.

FRIDGE does not create assembly code, the back-end of the C compiler is responsible for generating efficient code for the fixed-point operations. Relying on an external C compiler makes FRIDGE portable to any architecture, but will probably generate sub-optimal code. In order to encode the fixed-point operations efficiently the C compiler must recognise common patterns in the output C program and generate appropriate code sequences.

Simulation Approach The R2D2 project investigates the automatic synthesis of components in the context of reconfigurable computing. Menard et al. [46] analyse programs written with floating-point operations. The dynamic range of each variable is analysed using the same algorithm as FRIDGE [81]. In order to compile the program for a DSP there are several possible ways to schedule the instructions. Not all of the schedules compute the same bit-exact result. Menard et alùse an SQNR metric as a constraint for which schedules are acceptable. Program execution time is modelled as the product of each instruction length in the program, and the number of executions. The execution time is minimised under the SQNR constraint.

Autoscaler [37] is a source-to-source approach that operates on ANSI-C. The source program is automatically annotated with instrumentation in SUIF [44]. For each variable the maximum absolute value, the sum of the values, the sum of the value squared, and the number of assignments is recorded. A conservative analytical method is used to compute the upper bound for each variable integer bit width. Standard optimisation algorithms are used on the recorded data distribution to minimise the integer bit width without causing overflow. The simulation approach is only safe when the input signal covers the run-time data. A model of the signal is required, but one advantage is that the model can be non-linear.

Hybrid Approach Cmar et al. [12] propose a hybrid of simulation and analysis within the context of the Ocapi project. The most significant bit position of a value is determined by propagating range information between variables. This produces a conservative result to prevent overflows. The least significant bit position is determined through a simulation approach. The simulation compares the floating-point values with fixed-point representations that introduce varying levels of quantisation error. Control-flow is decided during simulation by operations on the fixed-point values to ensure that both simulations proceed uniformly. The system is interactive, the error metric is computed for a specific quantisation of the system, and the designer must indicate whether this is intentional or not. Safe bounds are initially computed by the system. The designer can refine these bounds using the error information generated by the system until a suitably precise solution is found.

2.3.2 Instruction Set Architecture Support

There has been work in using Precision Analysis to support efficient compilation of code to specific Instruction Set Architectures:

Emdedded ISA Support Aamodt et al. [1] describe a new instruction set architecture that explicitly supports fixed-point formats through the introduction of a new instruction. The Fractional Multiplication with internal Left-Shift (FMLS) reduces the rounding noise and enhances run-time performance by allowing a more direct representation of a fixed-point algorithm in an embedded ISA. This new instruction has an associated algorithm, Intermediate-Result-Profiling based

Shift Absorption (IRA-SP), which discards internal most-significant-bits that are redundant because of inter-operand correlations.

Using SIMD Instruction Sets For Multimedia Code Pollard et al. [55] investigate compilation of multimedia code onto Single Instruction Multiple Data (SIMD) architectures. SIMD instructions operate on multiple pieces of data in parallel, using single instructions. The SIMD register set in mainstream processor designs uses integer operations on fixed size words. Pollard et al. use interval arithmetic to formalise the variable bounds in the program, and automatically pack the values into SIMD registers. Simply propagating the intervals through the program causes a widening of the intervals that each variable lies on. The paper investigates optimisation techniques to reduce this widening in the case where variables are reused within individual expressions.

2.3.3 Hardware Synthesis

Both the PICO NPA design system [67] and BitValue [9] independently developed bit inference algorithms to analyse which bits in a computation are necessary. The target domain is reconfigurable computing where each bit that can be eliminated has an impact on the size of the synthesized design. Both systems use a data-flow analysis with a forward and backwards part that iterates until a fixpoint of the system is reached. The forward part propagates *def* information — which bits are defined by operations — and the backwards part propagates *use* information — which bits are needed by operations. The *use* information is applied to remove unnecessary bits from the computation, which then effects the propagation of *def* through the code. Both analyses are iterated until they converge.

In the implementation of PICO each variable is represented by the number of bits (n) need to represent the mantissa. These values are assumed to be integer so the bitstring represented is from bits 0 to $n-1$. BitValue uses the more precise representation of a bitstring where each bit independently takes a value from $\{0, 1, u, x\}$. The values 0 and 1 perform constant propagation at the bit-level within values, u represents an unknown and x does not affect the output.

Bitwidth analysis of floating-point variables is studied in [22] for hardware synthesis. Gaffar et al. are motivated by reducing the silicon area of designs by eliminating individual bits within variables and thus the circuitry that implements operations between them. In order to safely reduce the bitwidth of values, the user supplies a cost function which specifies the maximum error in outputed results and a bound on the design area. The analysis determines the size of errors that are propagated through the program data-flow graph by reductions in the size of a variable. This process is termed sensitivity (or precision) analysis and is used to decide how the bitwidth of variables can be reduced. The analysis proceeds in two passes; firstly differentiating the transfer functions in the data-flow graph, and secondly propagating reductions in bitwidth backwards through the graph. The second pass is iteratively applied until the cost function is met. In order to propagate the errors back through the data-flow graph the possible error on each output node has to be partitioned amongst the input nodes. Two methods are presented to perform this task; either a uniform distribution or weighted by the area cost of each node.

In [23] Gaffar et al. extend their analysis to unify both floating-point and fixed-point arithmetic. Range analysis is used to determine the cost of representing values in each format. For values with a large dynamic range the cost of floating-point operations forms a trade-off with the reduction in the number of bits required to store values. FIR filters and ray-tracing are used as sample applications to demonstrate the technique. Further inputs from the user include a sample data set for simulation. The tool can then be used to explore the trade-offs between error tolerance, silicon area, dynamic range and the selection of which format to use. The representation of errors in BitSizer is predefined — IEEE double precision by default.

Recently Rugina and Rinard [66] demonstrated an alternative approach to the problem. They used a *constraint-programming* approach to perform an analysis that produces symbolic bounds on variables within the program. These bounds can be computed numerically when enough information is available. The analysis can be applied to pointer ranges, memory accessed, array indices and the integer case of the bitwidth problem. The class of programs considered are recursive *divide and conquer* algorithms that are hard to analyse using an approach that iterates to a fixpoint, such as abstract interpretation or a data-flow analysis.

2.3.4 Comparison of Previous Analyses To Our Technique

All of the work surveyed in Precision Analysis focuses on the problem of statically determining the size of representations for variables within programs. Part II of this thesis presents a language for describing FIR filters. The analysis in Chapter 6, and the compilation technique in Chapter 7 together form an automated transformation from this language into the executable language of the PIC microcontroller. Our analysis techniques uses constraint logic programming to find a solution for the variable sizes, and then applies a novel compilation technique to generate executable code.

Part I

Timing Semantics

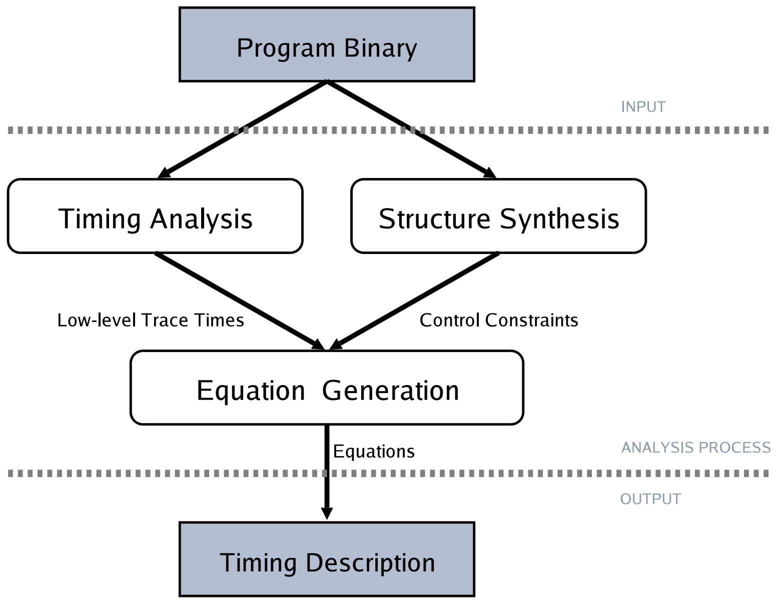


Figure 2.1: The Timing Analysis Process

Our process for analysing timing in programs is split into three separate stages. The data-flows between these stages of the process are shown in Figure 2.1. Stage I (in Chapter 3) is Timing Analysis, which operates directly on the program binary to produce the local timing solutions. These are the individual timing solutions contained within the fixpoint for the abstract interpretation. Each contains a pair of locations within the program and a number of clock-cycles. The number is the execution time for control flow to pass from the first location to the second.

The abstract interpretation in Stage I uses a gross assumption of control-flow to compute local solutions. Global timing solutions that rely on this coarse approximation alone will lack precision. Without more information to refine this approximation the global timing solution would diverge completely; each location would be executable at any time, giving no information about the timing of the program. The information that we use to refine this solution is the higher-level structure within the program. This structure defines loops in the program, the relations between them, and the scope of each loop.

Stage II (in Chapter 4) is the Structural Synthesis process, which computes this structural information directly on the program binary. This process is similar to reverse compilation, however in our problem domain there is no higher-level source code and thus we are synthesising a structure, rather than recovering one.

The final stage in the analysis process is to generate the timing set for each location. Stage III (in Chapter 4) of the process is Equation Generation. This process uses the local solutions from Stage I, and the structural information from Stage II. Both inputs are combined to produce a timing equation for each location in the program. The equation generates the set of possible execution times.

Chapter 3

Analysing Timing of Target Code

In its simplest form, a program is a sequence of operations to be interpreted mechanically. The program can input values, perform operations upon its internal state, and output results. The program defines a function over the input values onto the output values, which may be undefined for some input values. Certain effects may be observed from the program execution. The result output is defined as the *functional effect* of the program. There are other effects that can be observed, but do not alter the output value. We term these types of effects as *side-effects*. The side-effects produced by program execution are defined by both the structure of the program, and the operation of the machine. There are two methods for analysing the effects, and side-effects, of programs:

Dynamic methods execute the program with input values and observe the result. The mapping from input values to output values determines the functional effect. Instrumenting the program with measurements of non-functional state determines side-effects. To obtain complete results, in either case, a suitable set of test values for input must be selected that ensures all possible program executions are measured by the instrumentation. Selecting such a coverage for arbitrary programs is a hard problem in program testing, as it is not known which inputs could cause non-termination of the program under test.

Static methods execute an approximation of the program over all possible input values. This approximation is necessary to make the execution tractable. There is a necessary loss in precision in approximating the program, so static analyses perform a conservative approximation. This ensures that whilst some extra solutions to the problem are produced, no solutions are missed. As no test input is required for static methods they can be performed at compile-time rather than run-time. Dynamic methods can only be performed at run-time.

A side-effect of program execution is the mapping of the sequence of operations onto time. There is a dynamic method, called *profiling* for measuring this side-effect that does not use an analysis. The program is instrumented in order to measure the current time during execution against a set of test data. The differences between the time measurements show the length of time to execute individual parts of the program. As noted above, to obtain the complete set of program timings test data must comprehensively cover all execution paths (*traces*) within the program.

Timing Analysis is a static method for determining the timing side-effect. In WCET Analysis, described in Section 2.2, a program and a description of the machine are mapped onto a single value in the time domain — an upper bound on the length of time that the program execution will take. The class of target programs on which WCET can operate is restricted to *transformational* programs — input occurs before execution commences, and output is produced upon termination.

The terminology of reactive and transformational systems is taken from Pnueli's survey [53] of temporal logic in reactive systems. *Reactive* systems are categorised as holding an ongoing dialogue with their environment. This dialogue is considered to be infinite in length as reactive systems do not terminate. This definition is more relaxed than a transformational system, on when input and output occur, in which input is fixed at the start of execution and output fixed at the end.

Because input/output actions are not restricted to the beginning and end of execution, WCET techniques are not directly applicable on this class of program. The observable actions whose timing is being analysed are frequently located within loop nests in target programs. The times that are required are not between neat scope boundaries in the program and may involve partial execution of some scopes. Furthermore separate actions may occur at different frequencies within a system, creating a change in phase between them. An analysis that assumes execution of complete blocks of code cannot determine these timing relationships.

We present a new static analysis that generates a model of the temporal behaviour of a reactive component. The analysis operates directly on machine code, making it applicable to both legacy code and compiler output. Our Timing Analysis differs from previous work in two ways.

- The programs under analysis are reactive
- Exact times are produced, rather than upper-bounds.

The Timing Analysis produces a model of the times at which each instruction in the program can execute. This model allows verification that the externally observable events within the program can only occur within a valid set of times.

The problem domain that the Timing Analysis has been applied to is the design of wearable computers, described in Section 1.1. The simplicity of the PIC micro-controllers in the system necessitates a time-triggered programming model. There is no support for fine-grained event triggers from external hardware. In particular, the latency on the micro-controller using an interrupt to awaken from a fully suspended state is too high to allow a rapid sampling of the sensor hardware state. Instead a simple polling model is used to supply data.

The problem domain and constraints on the device guarantee that the code between each interaction with the environment will execute a bounded finite computation. This separation of observable effects by bounded computations characterises the programs on each device as real-time processes. The real-time constraints on a process define when the interactions must occur, which is equivalent, in this domain, to defining the time taken to execute each computation. In this class of system the path between observable actions may have multiple possible traces, each trace must form a bounded computation, but the bounds in each trace may differ.

Given this class of programs the model produced by the Timing Analyser has been used in two ways:

Verification of legacy code. In model-checking the correct operation of the program is mechanically verified against a model. The specifications of such models are complex to define and use. Instead the analysis generates a model of the program's behaviour in time which can be checked by the programmer against their intuition of the correct behaviour.

Statically scheduling processes. In order to statically schedule processes in a real-time concurrent language it is necessary to construct a model of the temporal behaviour of each program fragment. These models are used to determine if the decisions to schedule processes are static and can be reduced out of the program at compile-time. One example application is statically splicing a real-time thread into a host thread at compile-time [17].

3.1 Example System

The timing analysis is motivated by legacy code on the wearable computer system described in Section 1.2. Devices within this system communicate over a shared bus that operates a simple teletype protocol. The devices must maintain communication on this bus while performing their main function. In the devices that operate PIC micro-controllers this requirement places a real-time constraint on the assembly language program. The serial transmission code for this teletype protocol is sufficiently small, and simple, to act as an example to perform the timing analysis upon. Throughout this chapter we will demonstrate the steps in the analysis on this sample as an aid to the reader.

Communication over the shared bus uses a simple teletype serial protocol. A serial protocol uses unidirectional lines that can be driven in one of two states: high and low. Such lines can be paired together to form a bi-directional bus. Some buses use extra lines to encode a third state for arbitration of access. For simplicity, in the example code and system we shall consider the case without arbitration. Using the shared line to communicate data requires some agreement between the parties on when a sample of the line state forms a data-bit. The teletype protocol is asynchronous — using a bus without a clock requires fewer components, which is desirable in a wearable system.

As there is no shared clock to synchronise sampling of the bus, each end maintains a local clock. These clocks are resynchronised on each transmission and merely have to remain within a set margin of error during the transmission. In order to observe the beginning of a transmission correctly the two ends must agree to a constant baud rate B. When the line is idle between communications the transmitter holds it in the high state. On each transmission the line is driven into the low state for $\frac{1}{B}$ seconds. This marks the beginning of the transmission and allows the far end to synchronise its clock. After the start bit follows n data-bits, for which the line is held in a state for $\frac{1}{B}$ seconds each. For each bit the high state signals a 0, and the low state a 1. Finally, the line is driven high for $\frac{1}{B}$ seconds.

The parts of the transmission are called the start-bit, data-bits, and the stop-bit. Each party must agree to the same variation of this format. The sample code operates a 1-8-1 format, that is 1 start-bit, 8 data-bits and 1 stop-bit. The receiver samples the line until it observes a low state long enough to constitute a start-bit (this avoids extraneous transmissions caused by line-noise). The line is then sampled at times $\frac{1}{2B} + \frac{n}{B}$ for $0..n$-1 data-bits. This captures the line state in the middle of each bit transmission. The process will correctly communicate the data-bits (in the absence of line-noise) if the drift between the two clocks is less than $\frac{1}{2B}$ seconds over the $\frac{10}{B}$ second transmission time, or 5%.

The example program, shown in Figure 3.1, implements the teletype protocol. The language is the instruction set of the PIC micro-controller. This was necessary for the legacy code as there was no suitable higher level language that could express the timing properties. Languages that allow timing bounds over program constructs, such as Esteral, are not capable of cycle-accurate instruction placement. The program uses a technique known as *bit-banging* — a busy loop that manipulates the hardware registers for the serial line. The loop is busy as it occupies the micro-controller, padding the idle time in the loop with NOP instructions.

```
0    XMIT    MOVWF    SER_TX       ; Data shifter
1            MOVLW    NUMBIT+1     ; Byte size + start bit
2            MOVWF    BITCNT       ; Preset data counter
; Send the start bit
3            BCF      PORTB,TXD    ; Set start bit level
4            GOTO     XMITC        ; Wait for start element
; Set the transmit data level from the carry and wait for the correct length of time
5    XMITA   RRF      SER_TX,1     ; Shift register right
6            BTFSC    3,0          ; through carry
7            GOTO     XMITB
8            BCF      PORTB,TXD    ; Data is '0'
9            GOTO     XMITC
10   XMITB   BSF      PORTB,TXD    ; Data is '1'
11   XMITC   NOP
     ; call WAITEM ; Wait for the element to go
             ;NOP
12           NOP
13           NOP
14           NOP
15           NOP
16           NOP
17           NOP
18           NOP
19           NOP
20           NOP
; Count the elements as they are sent
21           MOVF     BITCNT,1
22           BTFSC    3,2
23           RETURN
24           DECFSZ   BITCNT,1
25           GOTO     XMITA
; Bit count has zeroed, send the stop bit
26           BSF      PORTB,TXD
27           GOTO     XMITC
```

Figure 3.1: Serial transmission example

These have no functional effect, but fix the time period of each loop iteration.

The main loop is XMITC which executes 10 times, once for each bit. The initialisation code jumps into the middle of this loop, rather than entering through the head in order to produce the right period between the instructions that control the hardware interface. The commented out sections are retained to document the trial and error that is required to produce the correct temporal execution. The example code operates the serial line at 115200 baud. The microcontroller uses a 10MHz crystal to operate at 2.5MIPS. This requires the line state to be changed every 21.7 clock cycles, the closest possible on a discrete clock is every 22 cycles which is within the 5% error tolerance – assuming that the crystals at each end agree on 10MHz closely enough. The example code demonstrates fine-grained timing synchronisation, as cycle-accurate instruction placement is necessary to operate the serial line at the correct rate.

The purpose of applying the timing analysis to this particular code is verification. The important locations within the program are the entry point, the instructions that alter the bus state, and the exit point. To execute correctly, each of the possible traces between these points must take exactly the correct number of clock cycles. In this particular case it is necessary to verify that all possible program paths between these points are of uniform time, and that the time is correct.

3.2 Instruction Set Model

Each processor has an instruction set that defines the format and operation of programs. In addition, each implementation of an instruction set defines the length of time taken, by each instruction to execute. These times are measured in the number of cycles of the processor clock. For the rest of this chapter we define a *time* to be an integer number of clock cycles. Conversion of these times (relative to a specific implementation clock) to an absolute time will require division by the frequency of the implementation clock.

A program is a mapping of locations onto instructions. Within each processor there is a program counter (PC) that stores the current location at each clock cycle.

Each instruction has a defined effect upon the program counter, which selects the next instruction for execution. The particular sequence of locations / instructions executed is called a *trace*. With a definition of time in the system, it is possible to define the length of a trace.

The PIC instruction set implementation is used for illustration in this chapter as the time of execution for each instruction is regular. It is a function purely of which instruction is executing, and not of the internal state of the processor. This property would not hold in a pipelined, or superscalar architecture. While the Timing Analysis could be applied to such an architecture, it would require the internal state of the processor to be held in the abstract states during analysis, making any exposition more detailed and complex.

Each instruction within the program is a single control point. During each discrete cycle of the processor clock one such point controls the processor as the instruction is executed. There is a unique label for each control-point termed a *location*. These labels are non-negative integers and the set of labels in a program is referred to as $L \subset \mathbb{Z}$. The program P is then a mapping from the set L to the set of instructions.

The processor contains a state comprised of the set of assignments of values to registers and flags. We will term this state S. The program counter is one of the registers in this state and thus the transfer function of the processor is a mapping on S, parameterised by the program. This transfer function is shown in Equation 3.1.

$$S' = f(P, S) \tag{3.1}$$

The transfer function encodes the semantics of the instruction set as changes in the processor state. These changes in state are broadly divided into three categories, and every instruction has effects in each of these categories:

Functional effects modify the contents of registers or flags.

Temporal effects change the processor's clock. These effects are not strictly caused by the execution of an instruction, but are a side-effect of it.

Control flow effects modify the program counter, determining the next control point in the program.

There is an overlap between functional and control effects, as the PC is a register within the processor state.

The values that we operate upon in the concrete domain are times represented by integers; $t \in \mathcal{Z}$. The time t is not strictly part of the state. There is no explicit storage of the clock within the processor, instead the clock is observed externally as a visible side-effect of program execution. A *configuration* is a combination of a state (time) and the control-point that the state is active at within the program. Equation 3.1 can be made more detailed by explicitly exposing the temporal and control effects in the transfer function. Equation 3.2 shows this explicit transfer function operating on configurations. The processor state S is shown for completeness, although we do not consider it within the analysis.

$$[S', \text{PC'}, t'] = f(P, [S, \text{PC}, t]) \tag{3.2}$$

An abstraction simplifies structure by removing detail. All details of functional effects are removed in the Abstract Instruction Set. Program executions in this Abstract Instruction Set are independent of input data. The transfer function for executing abstract programs is shown in Equation 3.3 below. The processor state, except for the PC which is part of the configuration, has been abstracted away from the explicit transfer function. The abstract transfer function has multiple solutions for some locations within the program. When the control flow is dependent on functional effects, such as conditional branch instructions, there is a solution of the transfer function for each possible control-flow result. Under abstract execution of the program, control-flow transfers to a set of configurations, unlike concrete execution of the program in which control is transfered to a single configuration.

In the abstract domain each state is a set of possible times, rather than a single integer. The partial order used over states is $\in$, allowing definition of least upper-bounds in the lattice of states.

$$[\text{PC'}, T'] = f(P, [\text{PC}, T]) \tag{3.3}$$

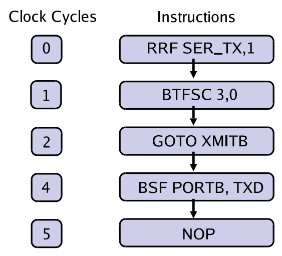


Figure 3.2: Concrete Trace

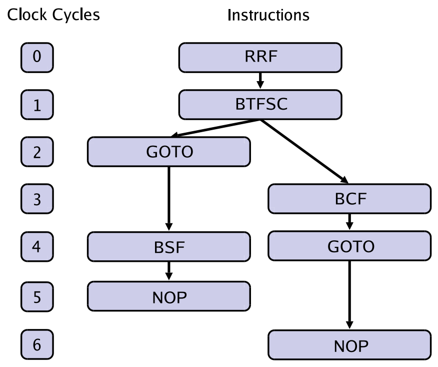


Figure 3.3: Abstract Trace

Traces of programs are sequences of instructions as shown in Figure 3.2. The trace of an abstract program is a tree, in which the children of each node are the set of possible successive instructions, as shown in Figure 3.3. The trace of an abstract program contains the set of all possible concrete traces which form the paths from the root of the tree to each leaf. Figure 3.2 & Figure 3.3 are traces of the same fragment of the example program. The concrete trace is the path from the root of the abstract trace to the left leaf. The times relative to the start of the trace are shown for illustration. As the instruction set takes different numbers of cycles to execute different types of instruction, there is a missing cycle for the concrete trace where an instruction takes two cycles. Each level in the tree shows the possible instructions that can execute in that cycle.

$$\gamma(T) = \{[s,t] : \forall s \in S, \forall t \in T\} \tag{3.4}$$

$$\alpha([s,t]) = [\{t\}] \tag{3.5}$$

Each of the abstract configurations generated in Equation 3.3 represents a set of concrete configurations in the program. As shown in Equation 3.4 the set of concrete states is over all possible processor states for the abstract state. The mapping from an abstract state to the set of concrete states is termed concretisation, and is conventionally denoted γ. The abstraction function is termed α and maps a concrete state onto a single abstract state as shown in Equation 3.5. The abstraction removes all state information from the program, only the PC remains, as in Equation 3.3.

To be a valid abstract interpretation the pair of functions α and γ must form a Galois Connection. The Galois Connection is a property of the compositions of a pair of functions. The required property is shown in Equations 3.6 & 3.7 and ensures that execution of an abstract program is a safe approximation of executing the concrete program. The safety property ensures that mapping a concrete state to the abstract domain and back again results in a set of states that contains the original. In the reverse direction the property ensures that when an abstract state is mapped to a concrete set and back, a superset of the original state is obtained.

$$\forall[s,t] : [s,t] \in \gamma(\alpha([s,t])) \tag{3.6}$$

$$\forall T : T \subseteq \alpha(\gamma(T)) \tag{3.7}$$

By considering only the temporal effects of each concrete instruction the complexity of the state operated upon by each abstract instruction is greatly reduced. Instead of analysing the contents of each register and control flag within the processor only a single integer representing the current time is required. Under this abstraction the number of unique instructions is reduced. The abstract instruction set consists of the unique temporal effects contained within the concrete instruction set. When multiple instructions have the same control and temporal effect, and differ only in functional effects, they are equivalent under this abstraction. Thus the abstract instruction set is smaller and simpler than the concrete instruction set.

In the case of the PIC micro-controller the 35 unique instructions form 5 equivalence classes under the timing abstraction. The Timing Analysis operates directly on these classes, which are termed the Abstract Instruction Set. In the PIC these classes are:

SINGLE Takes one cycle and passes control to the next address ($\text{PC} + 1$)

SKIP Takes either one or two cycles and passes control to either $\text{PC} + 1$ or $\text{PC} + 2$

$$\begin{array}{lcll}
(\textsc{pc}, t) & \Rightarrow & (\textsc{pc}+1, t+1) & \text{when } P(\textsc{pc}) = \textsc{Single} \\
(\textsc{pc}, t) & \Rightarrow & (-1, t+2) & \text{when } P(\textsc{pc}) = \textsc{Return} \\
(\textsc{pc}, t) & \Rightarrow & (\textsc{pc}+1, t+1), (\textsc{pc}+2, t+2) & \text{when } P(\textsc{pc}) = \textsc{Skip} \\
(\textsc{pc}, t) & \Rightarrow & (tar, t+2) & \text{when } P(\textsc{pc}) = \textsc{Jump}(\text{tar}) \\
(\textsc{pc}, t) & \Rightarrow & (\textsc{pc}+1, t+2+\text{analyse}(tar)) & \text{when } P(\textsc{pc}) = \textsc{Call}(\text{tar})
\end{array}$$

Figure 3.4: Abstract Instruction Semantics

JUMP (n) Takes two cycles and passes control to the address n encoded within the instruction

CALL (n) Takes two cycles plus the time to execute the target scope at n.

RETURN Takes two cycles and exits the current scope

The Timing Analysis is an abstract interpretation of the program transfer function. The abstract transfer function implements the semantics of the Abstract Instruction Set. The operational semantics of the transitions making up this transfer function are shown in Figure 3.4.

Conversion from PIC assembly code into the abstract instruction set is a simple many-to-one mapping for each location. Locations, code-lengths and therefore times are preserved when the program is represented in the abstract instruction format. The mapping of the sample program into the abstract instruction set, and then into a timing transition system, is shown in Figure 3.5. The transition system is created by substituting the transitions defining each instruction into the locations in the abstract program. The resulting transitions are between two locations in the program and take the specified length of time.

Analysis of the program occurs separately on each *scope* — defined as the reachable region from a particular entry point following the transitions defined by the SINGLE, SKIP, and JUMP instructions. When computing scopes the target of the CALL instruction is not included, only the transition to PC $+ 1$. Scopes are terminated by the RETURN instruction, by using a transition to a constant location outside of the program memory.

P(Sample Program)		Transitions applied to P(Sample Program)
n	$F(n)$	$(from, to, len)$
0	Single	(0,1,1)
1	Single	(1,2,1)
2	Single	(2,3,1)
3	Single	(3,4,1)
4	Jump(11)	(4,11,2)
5	Single	(5,6,1)
6	Skip	(6,7,1),(6,8,2)
7	Jump(10)	(7,10,2)
8	Single	(8,9,1)
9	Jump(11)	(9,11,2)
10	Single	(10,11,1)
11	Single	(11,12,1)
12	Single	(12,13,1)
13	Single	(13,14,1)
14	Single	(14,15,1)
15	Single	(15,16,1)
16	Single	(16,17,1)
17	Single	(17,18,1)
18	Single	(18,19,1)
19	Single	(19,20,1)
20	Single	(20,21,1)
21	Single	(21,22,1)
22	Skip	(22,23,1),(22,24,2)
23	Return	(23,-1,2)
24	Skip	(24,25,1),(24,26,2)
25	Jump(5)	(25,5,2)
26	Single	(26,27,1)
27	Jump(11)	(27,11,2)

Figure 3.5: Creation of Temporal Transition System

The target of the CALL instruction forms the entry point to a new scope. Each scope is analysed individually. We restrict the analysis to non-recursive systems of procedures, and hence the call-graph between scopes forms a tree. Analysis is performed on each scope from the leaves upward. This order of analysis ensures that when a scope is analysed the scopes that are called have already been analysed as a set of times. This set of resultant times for the scope at n is referred to as analyse(n) in the operational semantics for the transition function, described below.

On the PIC the SKIP instruction class is a set of instructions that test logical decisions and pass control along one of two transitions. It implements a guarded choice of the next instruction in the program that is used to build more complex control-flow structures such as guarded jumps. Abstracting the functional effects in the instruction sets makes these decisions non-deterministic. In the Abstract Instruction Set this is modelled by selecting both possible transitions.

When a program loops, it contains a jump to an instruction previously executed. This is the case in the outermost loop of the reactive program under analysis. There are also inner loops within these programs that are taken conditionally. In both cases an abstract interpretation of the program will fail to terminate, as it analyses traces of infinite length. We will prove that there is a failure to terminate, using abstract interpretation in our model. This termination property is invariant across any implementation, and not just limited to our implementation:

Theorem 3.1. *Abstract Interpretation will fail to terminate on the timing abstraction of any program containing a loop.*

Our proof of the non-termination theorem is quite simple and informal. Fixpoint detection cannot terminate in the presence of infinite ascending chains in the lattice of abstract states - if one of the states in the chain is present in the fixpoint. In order to prove non-termination it is sufficient to show that the lattice contains such chains, and that analysis will attempt to find a fixpoint containing a state on such a chain.

Proof. Consider two locations l_1 and l_2 which form a loop that is reachable in the program under analysis. l_2 contains a transition to l_1, and there exists a path (sequence of transitions) from l_1 to l_2. When abstract interpretation reaches a configuration at l_2 there are two possible states at l_1. If control has not yet passed l_1 then the state is $\oslash$ and so the transition from l_2 to l_1 will create a unique state in that configuration. In the case where control has already passed l_1 then the state at l_1 must contain a maximal time t_1. The latest execution of l_1 is at the current time t_2 (when $l_1 = l_2$), however all transitions contain non-zero length edges, and so the new state will be greater than t_2. As $t_1 \leq t_2$ the new state must be unique. As there is a path from l_1 to l_2 once this process has occurred it must occur again. Inductively the chain of states is of infinite length, or equivalently, traces of the abstract program will be infinite in length. $\square$

Finite termination of the abstract interpretation can only be guaranteed by the absence of infinite ascending chains in the lattice of states. The proof above shows inductively that infinite ascending chains exist in the lattice when loops exist in the program. As our problem domain consists of non-terminating looping programs this presents a problem. The solution is to use an upper-bounding operator on states within the lattice that guarantees all chains are of a finite length. We describe such an operator in Section 3.3.

3.3 Upper Bounding Operation

In Abstract Interpretation an Upper Bounding Operation is a second approximation. When the abstract domain cannot be computed, an approximation is taken of the abstract domain to produce a third domain. This approximation consists of a second pair of abstraction and concretisation function. The method is to repeat this approximation process over a set of increasingly abstract domains until the analysis is computable. The validity of the analysis is guaranteed over the transitive closure of approximations as long as each approximation forms a Galois Connection between the two domains [14].

This approximation process is shown in Figure 3.6 for the Timing Analysis. The pair of functions described in the preceding section are labelled α_1 and γ_1.

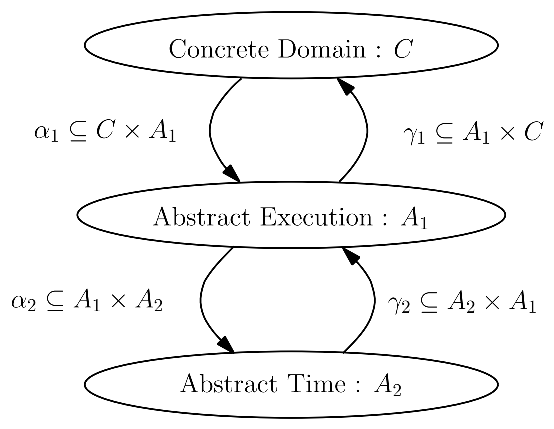

Concrete Domain : C
$\alpha_1 \subseteq C \times A_1$
$\gamma_1 \subseteq A_1 \times C$
Abstract Execution : A_1
$\alpha_2 \subseteq A_1 \times A_2$
$\gamma_2 \subseteq A_2 \times A_1$
Abstract Time : A_2

Figure 3.6: Operator overview in the Abstract Interpretation

They form a mapping between the concrete domain of programs and the first abstract domain; Abstract Execution. This domain abstracts the functional effects within the processor, and the control-flow within the program. In this section we construct the second abstract domain — Abstract Timing — and the mapping (α_2 & γ_2) between this domain and the domain of Abstract Execution.

In the Abstract Execution domain the values analysed at each program point are sets of times. These sets form a partial ordering under the $\subseteq$ operator. The lattice for this ordering is shown in Figure 3.7. The proof of Theorem 3.1 shows that this lattice is infinite in height, and that the state of all program points within a loop will ascend the lattice producing an unbounded set.

The infinite timing sets in the concrete domain are, by definition, non-repeating. However, only a finite portion of each chain contains new information for the analysis which is computing the possible lengths of time between pairs of program points. The transition system constructed for a program is complete; all traces of the program that can be executed are contained within an abstract trace of the program. This is a simple consequence of considering all possible transitions from an instruction under Abstract Execution. As the transition system for the program is interpreted the sets of concrete times computed are the reachable locations within a scope, and the length of time that passes before control reaches them. Once a loop is encountered new times are being generated, but they are copies of ear-

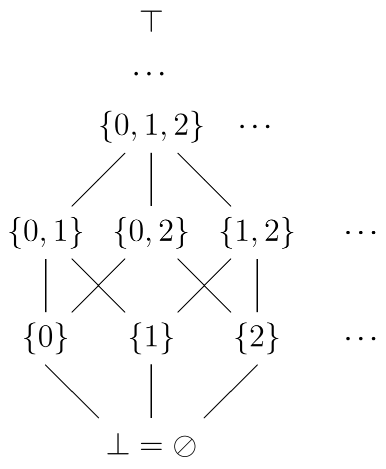

$\top$
$\cdots$
$\{0,1,2\}$ $\cdots$
$\{0,1\}$ $\{0,2\}$ $\{1,2\}$ $\cdots$
$\{0\}$ $\{1\}$ $\{2\}$ $\cdots$
$\bot = \emptyset$

Figure 3.7: Lattice of Timing States

lier times phase-shifted by the duration of the loop. The lengths of time between program points is limited by the number of unique paths through the program — which is finite for each program.

The upper bounding operation must fold the infinite set of timing states onto a finite set. Timing states from each iteration of a loop must be folded onto a smaller set of iterations. The simplest such set is the states in a single iteration of a loop. Our method of folding states is to measure timing within the program using a relative clock. Unlike the absolute clock in the Abstract Execution domain, this clock measures the time since a *recording point* was executed. The clock is reset when the recording point is executed again; as long as there are sufficient recording points in the program this guarantees that sets of timing states cannot increase without bound. As loops are executed in the program the timing distance from the initial program point increases, but within each iteration the relative timing distance from a constant point to each point in the loop remains constant.

An arbitrary choice of recording points will not reduce the height of the lattice. In order to ensure that the lattice is of finite height each loop in the program must contain at least one recording point. Our solution is to use an edge that is unique to each loop as the recording point for that loop as shown in Definition 3.1. For clarity we shall assume that such a set of unique edges has been computed for the program graph. In Section 4.3 we present an algorithm that computes the set of minimal cycles on the graph, these minimal cycles contain the unique edges required. An intuitively simple over-approximation of this set is the set of all jumps to earlier locations within the program.

Definition 3.1. *A* **recording point** *is an edge that is only contained within one loop in the progam. These edges are chosen in such a way that every loop contains at least one recording point.*

The choice of an edge that is uniquely within each loop creates two important properties in the abstract domain. Each iteration of the loop *must* pass the recording point at least once. Each instruction in the loop is reachable from the recording point *without* passing the recording point of another loop. The longest trace of instructions that does not include a recording point is bounded by the longest program path that does not contain a loop, or conservatively, by the size of the program.

The infinite lattice in the Abstract Execution domain has been folded into a lattice where each chain has less than n elements, for a program with n instructions. Furthermore, in the PIC architecture the largest/highest state in the lattice contains $n/2$ states. This is a property of how many unique traces there are through the most timing divergent sequence for a particular architecture. In the PIC architecture it tends towards 0.5 traces per instruction.

In both the concrete domain and A_1 the timing states are single integers. In A_2 these timing states are tuples of integers (t, l). Each tuple contains the location of the current recording point l, always the last recording point executed in this program trace. The time t is the number of clock cycles since the recording point was executed. Initially the recording point is set to -1; a nominated location outside of the scope. When a recording point is executed, it becomes the current recording point within the state, and the clock is reset to 0 cycles.

The definition of the abstraction between A_1 and A_2 allows the construction of a transfer function for programs that are mapped into A_2. This function, defining the semantics of the program under analysis in A_2, is shown in Figure 3.8.

$$\begin{array}{lcll}
(pc,t,l) & \Rightarrow & (pc+1,t+1,l) & \text{when } P(pc) = \textsc{Single} \\
(pc,t,l) & \Rightarrow & (-1,t+2,l) & \text{when } P(pc) = \textsc{Return} \\
(pc,t,l) & \Rightarrow & (pc+1,t+1,l),(pc+2,t+2,l) & \text{when } P(pc) = \textsc{Skip} \\
(pc,t,l) & \Rightarrow & (-1,t+2,l) & \text{when } P(pc) = \textsc{Call}(\text{tar}) \\
(pc,t,l) & \Rightarrow & (tar,2,pc) & \text{when } P(pc) = \textsc{Jump}(\text{tar}) \wedge \\
 & & & \text{L(pc)} \\
(pc,t,l) & \Rightarrow & (tar,t+2,l) & \text{when } P(pc) = \textsc{Jump}(\text{tar}) \wedge \\
 & & & \neg\ \text{L(pc)}
\end{array}$$

Figure 3.8: Abstract Instruction Semantics On Abstract Time

The function is defined as a set of transitions on abstract configurations. For simplicity, each tuple within a state forms a separate configuration, rather than a program point and a timing set. The transfer function therefore operates on tuples of (pc, t, l), where each timing state is a set of the (t, l) tuples for a common pc. Two auxiliary functions encode the structure of the program under analysis.

$P : L \rightarrow A_1$ maps the locations in the program onto the set of abstract instructions.

$L : N \rightarrow Boolean$ decides if a location is contained in the set of loop-nodes.

Figure 3.8 shows the definition of the transfer function in the Abstract Timing domain. The use of recording points, according to the definition of loop nodes, allows us to prove the existence of a least fixed-point for this function. This proof uses the Cousot and Cousot proof [15] of the Tarksi fixed-point theorem [73]. Their constructive formulation of the proof relaxes the conditions in Tarski's original to show that any monotone function on a finite lattice has a least-fixed point. Furthermore, this fixed-point can be reached from below, starting at an initial state for the system and incrementally computing the transfer function until it converges.

Theorem 3.2. *All ascending chains in the lattice A_2 are finite in length (lattice A_2 has the Ascending Chains Property).*

Proof. The finite height of the lattice follows directly from the definition of a recording point. Each recording point is an edge such that every iteration of the loop must include the recording point. Therefore for every iteration of each loop in the program, at least one recording point must be passed. The semantics of the transfer function reset the clock to two cycles on each pass of the recording point. The clock element of the state cannot increase without bound. Therefore each element in the state is bounded, and the lattice must be finite in height. □

Theorem 3.3. *The transfer function on A_2 has a least fixpoint.*

Proof. A function that is both finite and monotone has a least fixpoint [15]. The finiteness of the transfer functions follows directly from Theorem 3.2. To prove the existence of a least fixpoint we simply need to show monotonicity. The Abstract Interpretation computes the transitive closure of the states from an initial configuration. The monotonicity of the transitive closure operator over set union proves our result. □

In abstract interpretation it is conventional to firstly define the pair of maps (α_2 & γ_2), and then use these maps to define the abstract transition system. In this case the map definition is complicated by each map using global knowledge of the program structure to transfer values. To avoid operating on these complex definitions we have defined the transfer function directly, and proven the existence of a least-fixed point for the system. For completeness we now present a denotational definition of the maps α_2 & γ_2 in Equations 3.9 and 3.9.

$$\alpha_2([l_0, T]) = [l_0, \{(l_1, t) : \exists(l_1 \overset{t}{\Rightarrow} l_0), t + \beta(l_1) \in T, l_1 \in L\}] \tag{3.8}$$

$$\gamma_2([l_0, T]) = [l_0, \{\beta(l_1) + t : \forall l_1 \in L, \forall(l_1 \overset{t}{\Rightarrow} l_0)\}] \tag{3.9}$$

Normally to show the correctness of the abstraction it would be necessary to prove a Galois connection between α_2 and γ_2. A Galois connection would show that ordering is preserved when mapping between the two domains — a necessary condition for a correct analysis. Composing α_2 and γ_2 in either order produces an *equality*; there is no loss of precision when applying them. The only loss of precision is caused by abstracting the control-flow in α_1. The transformations α_2 and γ_2 locally preserve semantics; they only form an abstraction when composed with α_1 and γ_2. In Chapter 4 we will show how the abstract states of A_2 can be used to construct expressions that exactly generate the timing states of A_1. As this isomorphism between domains is strictly stronger than a Galois connection we conclude that the Timing Analysis correctly maps semantics.

For each configuration of (a, c, a') we apply the relevant transitions from the transition system to produce a set of new configurations. These are the transitions where $f = a$. When the edge (f, t, l) that control is being transferred over, is a loop back-edge we reset the recording point a' to a and reset the time to 0. Where f is the original location, t is the new location, and l is the number of clock cycles that the instruction occupies. Each new configuration is therefore $(t, c + l, a')$ when not resetting the recording point, and (t, l, a) when resetting the recording point.

The union is calculated of the current set of configurations and the set of unique new configurations, created by applying the transitions to the current set. This union is computed until a fixed-point is reached. The existence of this fixed-point is guaranteed by the finite number of locations and possible states.

This fixed-point contains the set of all time distances from recording points (the program start point and the set of loop back-edges) to program locations. This result can be seen as a reachability distance, showing not only which locations are reachable, but also the distance in time to reach that location.

3.4 Results on Example Program

We have implemented the Abstraction Interpretation in Haskell and performed experiments on program code to validate its performance on realistic code. Applying the analysis to our sample program gives several useful timings which we can use to verify the correctness of its execution. These are the period of the main loop executed for each bit, and the phase of the bit setting operations within this loop.

The fixed-point of configurations for the sample program is shown in Figure 3.9. We have highlighted the relevant configurations within this set to illustrate the correctness of the program with respect to the timing criteria that we now set out. The actions with temporal properties that we wish to verify are carried out by the target instructions within the program. The configurations that are relevant show the time distances between the back-edge of the transmission loop and the target instructions.

There are three temporal properties of the bit transmission loop that we verified. These properties all concern the main transmission loop. We wish to ensure that it has a uniform period, that the period is correct, and that the phase of the observable actions within this loop are correct and constant.

3.4.1 Uniform Loop Period

The main transmission loop (with back-edge at location 25) should have the same period upon each iteration. The set of configurations that form the fixpoint of the timing behaviour for the program contains the periods of each loop within the program. These periods are encoded as timings from the back-edge of a loop to the same back-edge node. We can extract this set through a filtering operation upon the fixpoint set. This filtering operation maps one set of configurations onto a second set of configurations according to a boolean operator on configurations. Filtering sets in this way using a higher-order function is a common idiom within functional languages and so we present the pseudo code for this operation in the syntax of Haskell. If this set of loop periods is a singleton set then we have verified that the loop has a uniform period across all executions of the program.

```
loopTimes :: Set Configurations -> Int -> Set Configurations
loopTimes fp loop = filterSet cond fp
   where cond (l,f,t) = (l==loop && f==loop)

uniform :: Int -> Set Configurations -> Boolean
uniform loop fp = 1==setSize (loopTimes loop fp)
```

((-1),14,9)	((-1),14,27)	((-1),18,0)	((-1),19,25)	(0,0,0)	(1,1,0)
(2,2,0)	(3,3,0)	(4,4,0)	(5,2,25)	(6,3,25)	(7,4,25)
(8,5,25)	(9,6,25)	**(10,6,25)**	(11,2,9)	(11,2,27)	(11,6,0)
(11,7,25)	(12,3,9)	(12,3,27)	(12,7,0)	(12,8,25)	(13,4,9)
(13,4,27)	(13,8,0)	(13,9,25)	(14,5,9)	(14,5,27)	(14,9,0)
(14,10,25)	(15,6,9)	(15,6,27)	(15,10,0)	(15,11,25)	(16,7,9)
(16,7,27)	(16,11,0)	(16,12,25)	(17,8,9)	(17,8,27)	(17,12,0)
(17,13,25)	(18,9,9)	(18,9,27)	(18,13,0)	(18,14,25)	(19,10,9)
(19,10,27)	(19,14,0)	(19,15,25)	(20,11,9)	(20,11,27)	(20,15,0)
(20,16,25)	(21,12,9)	(21,12,27)	(21,16,0)	(21,17,25)	(22,13,9)
(22,13,27)	(22,17,0)	(22,18,25)	(23,14,9)	(23,14,27)	(23,18,0)
(23,19,25)	(24,15,9)	(24,15,27)	(24,19,0)	(24,20,25)	(25,16,9)
(25,16,27)	(25,20,0)	**(25,21,25)**	(26,17,9)	(26,17,27)	(26,21,0)
(26,22,25)	(27,18,9)	(27,18,27)	(27,22,0)	(27,23,25)	

Figure 3.9: Fixed-point for sample program

The filtering expression (`filterSet cond fp`) applied to the fixpoint of the example program, yields the result $\{(25, 21, 25)\}$ — a singleton set. Hence the expression for the example program (`uniform 25 progfp`) is true proving that the main transmission loop has a uniform period.

3.4.2 Correct Loop Period

It is a common requirement of time-triggered reactive programs that we must verify the period of their loops which contain interactions with the environment. In order to perform this verification we perform the same filtering of the fixpoint as the previous section, and then map the configurations into the set of integer periods.

```
periods :: Set Configurations -> Int -> Set Int
periods fp loop = mapSet extr (loopTimes loop fp)
    where extr (l,f,t) = t
```

Performing this operation (`periods 25 fp`) on the example program yields the set $\{21\}$ which gives the period in processor cycles of the main transmission loop. This deviates from the target of 21.7 cycles and executing the program over 10 bits will take 3 cycles too few. This is within the specified tolerance and so the length of the loop period has been proven correct.

3.4.3 Constant Operation Phase

The bit manipulation instructions at locations 8 and 10 in the example program form an operation that interacts with the environment. These two instructions are on mutually exclusive program branches, so that exactly one of the two will execute on each iteration of the loop.

The distance in time from the execution of the back-edge of the loop to the visible operation is the phase of the operation within the loop. For some operations, such as the one shown in the example program, this phase should be constant in each iteration. In the example program a change in phase of the bus actuating operation would introduce a bias into the sampling of the line state.

In order to verify that an operation has constant phase we must ensure that the set of locations forming that operation all have constant distances from the loop back-edge in the program fixpoint. This is a similar filtering operation to the check for constant loop period. For a given set of instruction locations and a loop location we must compare the set of matching distances within the fixpoint.

```
phaseInst :: Set Configurations -> Int -> Int -> Set Int
phaseInst fp loop inst = mapSet mop (filterSet cond fp)
    where cond (l,f,t) = (l==inst && f==loop)
          mop  (l,f,t) = t

phaseOp :: Set Configurations -> Int -> Int -> Set Int
phaseOp fp ins loop = unionSets (mapSet phaseInst ins)

constantPh :: Set Configurations -> Set Int -> Int -> Boolean
constantPh fp ins loop = 1 == setSize (phaseOp fp ins loop)
```

The expression `constantPh fp {8,10} 25` evaluates to false, proving that the phase of the instructions differs causing a bias in the width of the pulses that is dependent upon the data being transmitted.

3.4.4 Correct Operation Phase

The last type of verification that we shall show is checking the actual phase of operations within the loop. In the example that we have presented this would be necessary in order to determine how much bias is introduced into the width of the pulses being transmitted. In other programs it may be necessary to check the phase of separate operations in order to verify a relationship between them. One example is sensors that transmit data encoded into pulse widths, modulated to a duty cycle. It would be necessary that the phase of operations relative to to duty cycle remained constant.

The code already presented is sufficient to determine the set of phases that an operation can occur at. The expression `phaseOp fp {8,10} 25` evaluates on the example program fixpoint to $\{5, 6\}$ showing that the phase of the operation differs by up to one cycle. This determines the amount of bias that this error in the program introduces.

3.5 Conclusions

This chapter described a new form of timing analysis that can be applied to the programs of non-terminating reactive components. The analysis calculates fine-grained timing information across the locations within the program, which can then be used to verify the correctness of a program's temporal behaviour. The analysis has been applied successfully to a microcontroller, and used upon a real-world example to verify the correctness of its timing behaviour.

For more complex programs the direct inspection of the fixpoint would not be feasible. In nested loop structures the timing distances do not take into account iterations of the inner loops. The analysis is dependent on information about the program structure that needs to be computed. In Chapter 4 we show the analysis necessary to apply the Timing Analysis to more complex programs, and how to represent the results to allow for loop nests.

The Timing Analysis can be seen as the first of three stages in a full timing analyser. These three stages are analogous to the stages in WCET. The second stage detects structural properties of the program, such as loop bounds and infeasible paths through the program. The third stage will combine this structural information with the low-level timing distances to create generating expressions for the execution times of locations within the program.

The problem of establishing loop bounds is orthogonal to the problem that we have solved, of deriving timing distances across program code. However, both need to be combined in order to generate precise expressions for the temporal behaviour of each instruction within a program. In the next Chapter we consider the problem of detecting structural properties within programs.

Chapter 4

Using Timing Analysis on Legacy Programs

The Timing Analysis described in Chapter 3 is the first stage in performing a full timing analysis of a program. The result of the initial stage is a set of timings, each element describing the execution within a local area of the program. The aim of analysing timing in a program is to find the possible times that each location can be executed within. The set of possible execution times for each location is a global timing solution; it depends on the entire program, not just the local area that the results of the Timing Analysis describe. Generating these global timing solutions requires a process for taking each piece of local timing information and computing its contribution. The combination of all contributing local times produces the global solution.

The local timing solution defined in Chapter 3 is a set of tuples. Each tuple (pc, t, l) contains a location in the program pc, a loop location l, and the number of clock cycles t for control to pass from l to pc. There are multiple tuples with identical pc and l when there are different possible execution paths between the pair of locations. For each loop there is at least one tuple for every reachable location in the program. A subset of the tuples contains the times of loop periods in the program. For each loop there are a set of tuples that contain the possible lengths of time to perform a single iteration of the loop. These tuples record

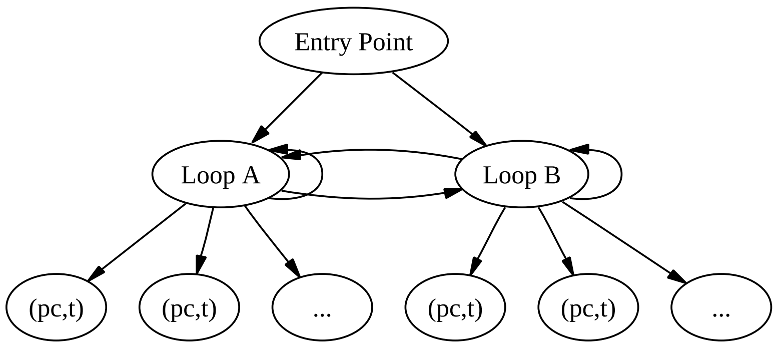


Figure 4.1: Structure in the Local Timing Solution

the time from each loop to itself in the form (pc, t, pc). The local solution also contains inter-loop timings of the form (l_1, t, l_2) and (l_2, t, l_1).

The structure of the local solution for a program with two loops is shown in Figure 4.1. A time is recorded from each loop to every reachable instruction. For programs with more loops the graph between loop vertices remains fully-connected within reachable regions. In the case that a loop is unreachable from another, some of the edges will be removed. This occurs in programs containing loops within initialisation code that passes control to a main system loop. Each edge between loops in the graph contains the set of times that can separate the execution of the instructions implementing the loop back-edge in the control-flow graph of the program.

An increase in program complexity will decrease the precision of the local timing solution. For each loop that is added to the program, another set of inter-loop timings is added to the solution. The time between iterations of a loop back-edge is not just dependent on the direct periods of the loop, given by (pc, t, l). Loops are also formed by cycles within the local structure graph. In Figure 4.1 control-flow can pass from Loop A to Loop B, and then back to Loop A. This adds the summation of the inter-loop edge times to the set of periods for Loop A. These cycles in the program structure are not limited to pairs of loops, but can include cycles of any size within the set of loop edges. Cycles are not limited to unique instances of a loop node within the graph, and can include forms such as:

Loop A $\Rightarrow$ Loop B $\Rightarrow$ Loop B $\Rightarrow$ ... $\Rightarrow$ Loop A

To convert relative program timings into an absolute form there is one more point that paths are timed from; the initial state. Each location in the program pc has at least one tuple in the local solution of the form $(pc, t, 0)$. The set of times measured between the initial state and a location gives the earliest possible time the location can execute. Equation 4.3 shows the computation of the timing set $T(l)$ for a location l using the local timing solution $F(l)$, and the set of loop locations $L(l)$.

$$I(l) = \{\, x \mid (l, x, 0) \in F\} \tag{4.1}$$

$$P(l) = \{\, x \mid (l, x, l) \in F\} \tag{4.2}$$

$$T(l) = \forall i \in I(l), \forall l \in L, \forall p \in P(l), \forall n \in \mathcal{N} \ : \ i + n \cdot p \tag{4.3}$$

The loss of precision is clearly seen in the use of universal quantifiers on each element of the timing equation. The conservative approximation of iteration indices with no loop bounds is $\mathcal{N}$. Information about the program structure is necessary to decide which of the possible cycles in the local structure graph are valid. Without any structure the approximation must be that all cycles are valid.

The role of the Structural Synthesis is to determine which of these sequences of loops are valid. The paths within the local timing solution that involve invalid loop sequences can be removed to produce a more accurate solution. It is not possible to decide exactly which sequences of loops are valid on general programs. Simply determining if each of the possible sequences was of finite length would solve the halting problem. As the halting problem is undecidable it is not possible to determine which sequences of loops are valid for a general program.

Our approach is a conservative approximation of the possible set of sequences within the program. The possible sequences are identified in a parametric form. The parameters are constraints on the number of iterations that each loop performs. If more constraints are known then fewer possible loop sequences are enumerated. These constraints on the structural elements of the program are used to reduce the number of possible paths through the unstructured graph. The application of the constraints on program structure, to the unstructured graph, is possible

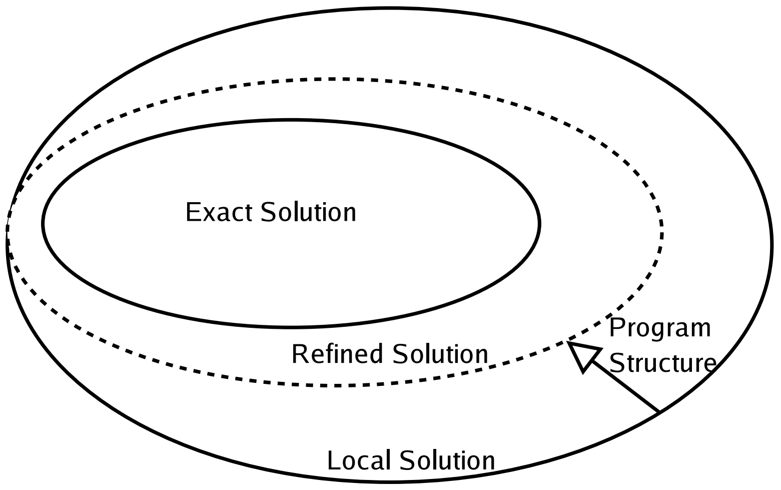


Figure 4.2: Relationship between Timing Solutions

because the synthesised structure forms a mapping between structural elements and the underlying sets of vertices that they relate to. The constraints remove some of the solutions in the local timing set.

The relation between the exact and refined solutions is shown in Figure 4.2. The exact timing solution is a subset of the local solution. The eliminated sequences in the local solution produce a refined solution. This refined solution is a subset of the local solution, but still a superset of the exact solution. The relative sizes of these three sets will depend on the specific structure of the program. It is not possible to guarantee that the refined solution will meet the exact solution in the general case. However, it is possible for specific programs to yield enough information to reduce the refined solution to equal the exact solution.

In order to constrain the timing solution, the structural synthesis computes the set of loops in the program, and their relationships with one another. The relationships are partial orderings on the set of loops. One ordering is called nesting, and corresponds to the hierarchy of loops in a structured program. When ordered in a nest, loops have inner- and outer-loops. The outer loops in a nest control the repetition of the inner loops in the nest. The other partial ordering is sequencing. When two loops are at the same level in a nest, there can be a sequencing dependency between them; that one loop must execute before another. These orderings are used to remove program paths that execute loops in an invalid sequence.

By computing the ordering relations on the loops in the program it is possible to generate program paths as a combination of sub-paths. The sub-paths are the local timings within particular loops, and the combination of sub-paths is controlled by structural constraints on the program. This definition of program paths, parameterised by structural information, allows many invalid paths to be removed from the local timing solution. In particular we can eliminate those paths that:

- Execute the loops out of order.
- Enter the loops within the nest in an order different to the hierarchy
- Exit a loop nest from an inner loop
- Include iterations of loops above known bounds
- Exit a loop before a known number of iterations

Computing the structural information to remove the desired paths from the timing solution requires the analysis of several parts of the program. In order to generate the parametric equations for the timing of each instruction, we must analyse the following information:

1. Identify the set of loops in the program
2. Identify a set of edges that can be cut to make the program graph acyclic
3. Compute the set of vertices within each loop
4. Fit a nesting hierarchy (partial ordering) onto the set of loops
5. Identify the sequencing constraints on loops
6. Place bounds on the iterations of each loop

In this chapter we tackle the first five analyses required. This provides sufficient information that the timing solution can be refined using any data available on the iteration constraints. We do not provide a method for computing the number of iterations of loops, although we note that in simple constant bounded cases a form of constant propagation should be sufficient, and that there is extensive work in the literature on more complex cases. In Section 4.1 we discuss the existing work on loop analysis and why it is not applicable to our problem domain. Then in Section 4.2 we describe our new definition of a loop that covers our problem domain. Section 4.3 gives the algorithms to implement structural synthesis using this definition. These techniques cover analyses 1,2,3,4 and 5 in the above list, and in Section 4.4 we show how to construct the global timing solution using these intermediate results. This timing solution is illustrated in Section 4.5 through application to an example program.

4.1 Previous Techniques for Loop Identifi cation

A loop is a simple concept in software design with an intuitive definition for a programmer — a series of operations in a program that are repeated. All non-trivial languages contain syntactic structures that can be used to create loops. In a declarative language the structure is recursion, allowing a function (or predicate) to be defined as a self-application. In procedural languages explicit structures allow a section of code to repeat while a condition holds. The underlying machine code representations of these structures use a JUMP to form a cycle in the control-flow graph of the program.

While it is simple to provide an intuitive definition of a loop, it is a complex problem to formally define what constitutes a loop. In analysing legacy programs there is no syntactic structure to mark the loops in a program. The machine code of the program is a set of instructions, each labelled with a location. This minimal representation forms a *control flow graph* for the program; each vertex is labelled with an instruction location, and each transition between instructions forms an edge within the graph. The problem of identifying loops is then a decision of

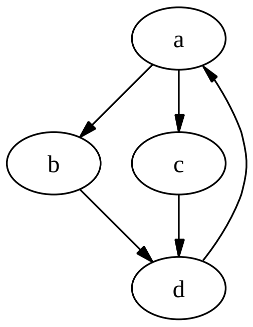


Figure 4.3: Strongly Connected Regions

which sub-graphs have the properties that form a loop. There are several possible definitions of a loop, which all identify different sets of sub-graphs as forming loops.

The simplest definition of a loop is a sub-graph that forms a strongly connected region. A *strongly connected region* is a set of vertices in the graph that are all reachable from each other. For each vertex to be reachable from the other vertices there must be a set of edges forming a cycle in the sub-graph. This definition is not suitable for finding the loops in a program. Consider Figure 4.3 which shows a control flow graph. The intuitive program structure in the graph is a single loop, which contains a conditional flow. But there are two strongly connected regions in the graph, $\{a, b, d\}$, and $\{a, c, d\}$.

A maximal strongly connected region is a *strongly connected component*. Such a region is maximal in that there are no vertices in the graph that can be added to the sub-graph whilst it remains a strongly connected region. In Figure 4.3 there is a single strongly connected component. The classical technique for finding strongly connected components in a flow graph is the Tarjan algorithm [71]. This efficient method of loop detection is an example of applying depth first search to a flow graph to produce a linear time algorithm for property detection. This is not suitable for finding the loops in a program as it will only identify the maximal, or outermost loops. A method using a similar technique that can identify inner loops is called *interval analysis*. Program loops will be detected by Interval Analysis provided that two conditions hold:

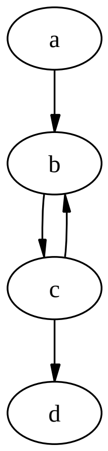


Figure 4.4: Single Entry Loop (reducible)

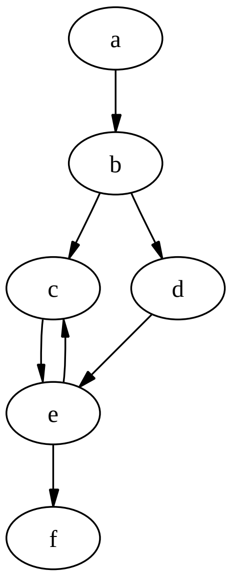


Figure 4.5: Multiple Entry Loop (irreducible)

1. The program flow graph must be reducible
2. Loops in the graph do not share headers

4.1.1 Graph Reducibility

When there is an edge from a vertex outside of a loop to a vertex within the loop, the vertex within the loop is known as an *entry point*. An entry point is a vertex that control can flow through before entering the cycle in the loop. Some loops have a single entry point, such as the example in Figure 4.4, while some have multiple possible entry points as shown in Figure 4.5. Interval analysis originated in methods for more general analysis of dataflow graphs [3]. The terminology *reducible* and *irreducible* applied to the types of flow graphs that were, or were not, amenable to these analyses. Later work showed that irreducible graphs contained loops with multiple entry points [72, 30].

Interval analysis operates on graphs that do not contain irreducible loops. For a reducible loop the entry point must form a *header* of the loop; a vertex through which control must pass in order to enter the loop. Although more efficient implementations exist, the simplest definition of an interval analysis is Hecht and Ullman's [30] T1-T2 formulation. Consider two transformations that can be applied to a flow-graph:

T1 If there is a single-node loop a, e.g. there is an edge $a \Rightarrow a$, then remove the edge from the graph.

T2 If a node a is the only node with an edge to a node b, then remove a from the graph, and modify all edges that did link to a so that they link to b. All duplicate edges are deleted.

A constraint on these two transformations is that the start node in a graph may never be removed. When a graph is reducible these transformations can be applied in any applicable order to result a graph containing only the start node. When a multiple entry loop exists there is no way to reduce it from the graph, even if the region reduces to a single vertex, application of T1 and T2 can never remove the multiple edges that link to it. The algorithm incrementally computes a sequence of graphs that reduce the original program graph to a single vertex. These reductions will always be performed 'inside-out'; inner loops will be reduced before outer loops. Thus a nesting hierarchy is computed for the loops as a side-effect while loops are identified.

4.1.2 Shared headers

Two examples of loops in a control-flow graph are shown in Figures 4.6 and Figure 4.7. In both figures multiple loops share a header. The *header* of a reducible loop is the entry point; the entry of a reducible loop must be the target of the loop's backward edge. Thus the vertex must be the first to be executed within the loop on each iteration. There is some disagreement in the literature about the meaning of a program that contains shared headers. T1-T2 Interval analysis is ambiguous in the presence of shared headers [82], depending on the ordering of the edges in the graph it may find a single loop, or two loops which can be nested in either way.

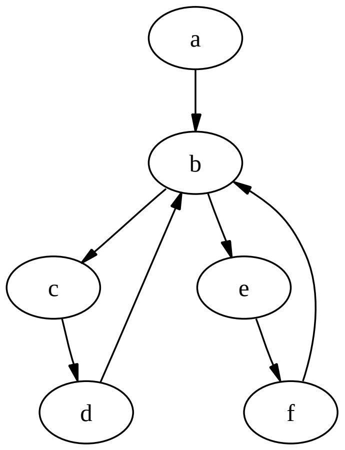


Figure 4.6: Shared Header - Same Loop

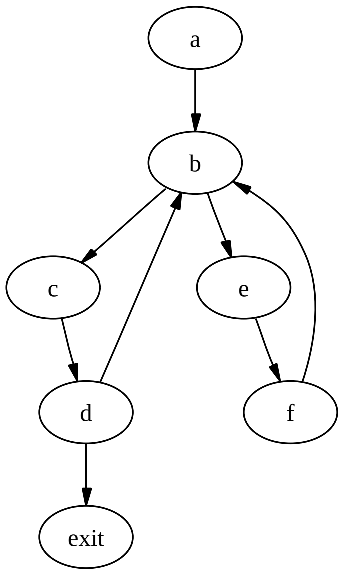


Figure 4.7: Shared Header - Nested Loops

Both shared headers and irreducibility are common properties in the control flow graphs of programs in the target problem area. When loop analysis is applied to the intermediate representation of a structured language, neither property occurs as high level language scopes are not overlapping. Either the scope of one structure is nested completely within the other or they are completely disjoint. The resource constraints of the target architecture make such structure an unaffordable luxury. The real-time deadlines are measured exactly in clock-cycles, and thus the paths between points in the code need an exact length. Replication of code to create a well-nested structure is not feasible because of the limited program memory available. In order to analyse the loop in programs of this type it is necessary to handle control flow graphs that can feature both irreducible loops and loops with shared headers.

4.1.3 Irreducible Loops

Of these two issues in loop analysis, more research has been devoted to analysing irreducible loops. One method of analysing irreducible loops is through a process of normalisation; extra vertices and edges are added to the graph. These vertices do not change the program semantics, but convert an irreducible region of a graph into a reducible region. Havlak presents one method [28] of normalising a graph that was originally thought to have linear time complexity, but was later shown to have a quadratic worst case [58]. The method modifies Tarjan's algorithm for reducible graphs by restricting the set of possible vertices in the graph sequence to descendants of the loop header in a depth-first-search. The identification of loop headers and back-edges depends on the exact depth-first-search chosen for the graph, they vary under different edge numberings. A secondary algorithm is supplied that splits the headers of reducible loops from the bodies of irreducible loops. This prevents the same header being chosen for both loops; the results of the algorithm are the same as defining shared loops headers to be a single loop.

Steensgaard proposed an ”outside-in” method for loop identification [70]. Firstly the algorithm applies Tarjan's strongly connected components algorithm to find the outer loops in a graph. For each loop the set of entry vertices is computed, for each entry vertex there is an edge from a vertex outside of the loop. Edges from a vertex within the loop to an entry vertex are identified as back-edges for the loop. The sets of back-edges for each loop are removed from the graph, and the algorithm repeats until the graph is acyclic. The algorithm handles irreducible graphs as each strongly connected region may contain multiple entries and back-edges. Any loops that share entries will be merged by the algorithm; nesting relations cannot be found on loops with shared headers. The definition of a loop employed does not differentiate between loops that share a header.

Ramalingam compares [58] several algorithms [28, 69, 70] that extend loop analysis to graphs containing irreducible regions. In each case Ramalingam demonstrates how to improve the efficiency of the algorithm, reducing the complexity to near linear time. Each algorithm produces a different nesting forest for certain input, and these differences are compared to identify which algorithm best captures the intended meaning of the control graph. Later work [59] extends the analysis to characterise both an axiomatic and a constructive definition of loop nesting forests. The three existing algorithms are shown to be specific parameterisations of the definition, and a novel parameterisation is constructed to compute nesting forests. Finally, the work illustrates how to transform irreducible graphs into acyclic graphs in a way that preserves a particular property on the graph. The method can be used to apply simple analyses of acyclic graph properties to more complex irreducible graphs. Our new definition and construction of a loop nesting forest is not a specific instance of Ramalingam's general definition of loops. Our definition of a loop allows the presence of separate loops in a flow-graph that share common vertices; these loops are neither disjoint, nor subsets of each other. These overlapping loops are ruled out by the *proper nesting property* of Ramalingam's definition. A strong similarity to our work is the comparison of entry vertices between loops to determine candidates for merging.

Dominator trees are well understood within the literature as they are used in many compiler optimisations [2]. They are also very efficient as they can be calculated in linear time on an arbitrary control-flow graph [4, 8]. The node that execution starts at is labelled the start node. A node a is said to dominate another node b (a dom b) iff all paths from the start node to b pass through a. One consequence of this definition is that all nodes trivially dominate themselves. This definition is extended to say, a strictly dominates b (a sdom b) iff a dom b and $a \neq b$.

Post-dominance can be seen as the dominance relation with respect to the exit (rather than the entry) on the reverse control flow graph [16] (CFG), this is the CFG in which the direction of all arcs has been reversed. A node a post-dominates another node b iff all paths from b to the exit pass through a.

Control dependence [20] is examined by Wolfe [82] as a method for computing loop nesting forests. The technique is applied to several types of flow-graph. Each graph is analysed using the dominance relation to detect back-edges, interval analysis and T1-T2 analysis. None of the flow-graphs used to compare the methods are irreducible, although one contains an overlapping pair of loops, and one example uses shared headers. The control dependence relation was the only method that correctly defined the nesting of the loops with shared headers. Control dependence is defined using post-dominance, and so can only be applied to flow-graphs that contain exits. The relation cannot be defined on non-terminating processes.

4.1.4 Application of Previous Work to The Problem Domain

Each of the algorithms discussed in the previous section uses a different definition of what constitutes a loop. The definition encompasses which set of vertices within the flow-graph form the body of the loop, and the relation on loops that defines a nesting forest. There is some commonality between the definitions that captures the intuition of what a loop should be. Each loop must contain at least one cycle; if it does not then execution cannot repeat entirely within the body of the loop. A cycle is a path on the graph that starts and ends at the same vertex. Another common aspect of a loop definition is that all the vertices within a loop should form a strongly connected region. This is a stronger condition than containing a cycle; as each strongly connected region *must* contain at least one cycle, but also, each vertex must be in at least one cycle in the subgraph of the loop.

Where the various loop detection algorithms differ is in the choice of which cycles are combined to form a loop. It is desirable for the set of loops, and the forest they form, to match programmers intentions on the graphs that they have created, and their intuition of structure on others. The algorithms that we have surveyed can be split into three groups, depending on how the graph is processed:

1. Using a dominance relation to define ordering from the start
2. Using a post-dominance relation to define ordering from the end
3. Using strongly connected components and a subset relation to define an ordering based on inclusion

The earlier research analysing loops as intervals [3, 72, 30] used the first approach. On reducible flowgraphs a dominance relation gives rise to a natural definition of a loop as the union of paths backward through the graph from a backedge that do not pass the *header* node. Each region contains a single header node and a single backedge. Extensions [28, 69, 59] of these earlier approaches weaken the loop definition to incorporate irreducible regions. Multiple headers and backedges are allowed within a loop, but a proper loop nesting must be retained. The nesting forest is defined by the subset relation, and the bodies of every pair of loops must either be disjoint, or one must be a subset of the other. This requirement prevents an irreducible region from containing multiple independent loops that share vertices — *such as the loops with shared headers that we wish to analyse*.

The second approach is used by control dependence relations, Wolfe's study of which has already been discussed to the effect that this approach can analyse loops with shared headers but can only be computed on terminating scopes.

The third approach is used by Steensgaard [70] and will merge together loops that share vertices if those loops all form entries for the strongly connected component. *This includes the loops with shared headers that motivate our study.*

Our solution is to propose a definition of loops that can be applied to the types of control-flow graph in the problem domain, and be used to determine presence of loops more precisely than previous work.

4.2 A New Defi nition Of Loops

Each program is represented by a digraph. Formally a *digraph* is a set of vertices V and a set of edges E wwritten as $< V, E >$ where $E \subseteq V \times V$. Each of the labelled vertices is a location in the program. The edges are the control flow transitions, so that an edge from a to b implies that after the execution of a, b can be the next location to be executed. This change in the current location is known as passing control, and the process of passing control through the vertices in the graph is known as the flow of control. The set of control-flow graphs does not contain any graphs with multiple edges between the same pair of vertices. There can be loops between a pair of vertices, but each control relationship between locations is a single edge on the graph.

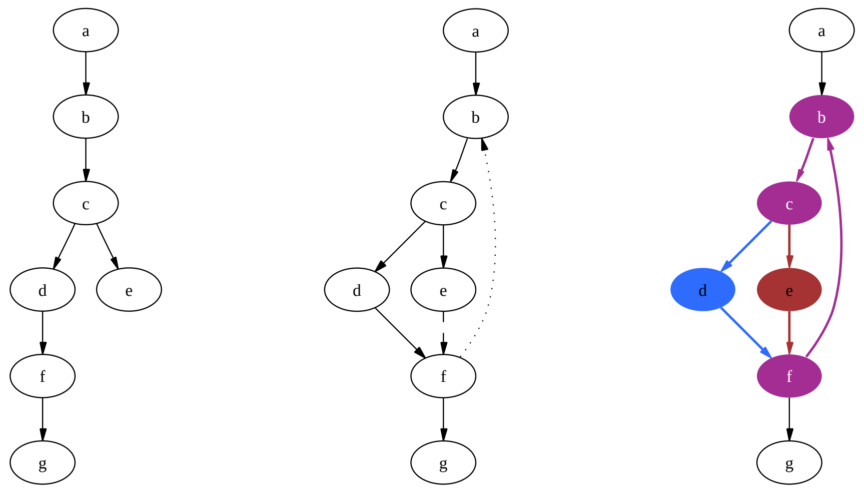
We will denote each edge tuple (v_1, v_2) in the edge set E by $v_1 \Rightarrow v_2$. A *path* is a sequence of vertices $[v_1, v_2, .., v_n]$ such that an edge links each adjacent pair of vertices in the correct order $\forall i \in \{1 .. n-1\} : v_i \Rightarrow v_{i+1}$. A path on the graph is a route that control flow can pass along between two locations. Each path on the graph is part of a trace of the program; an ordered list of the locations that are executed in an instance of running the program. In some places we shall refer to a path as containing edges. As there is exactly one edge between each adjacent pair of vertices in a path it should be clear which edge is intended.

A path that starts and ends on the same vertex is a *cycle*. We define a *minimal cycle* as a shortest path through an edge that forms a cycle. E.g. the minimal cycle defined by $a \Rightarrow b$ is the path $[v_1, v_2, .., v_n]$ where $v_1 = a$, $v_2 = b$, $v_n = a$ and $n \leq$ the length of any other cycle containing $a \Rightarrow b$. Not all edges in the graph lie on cycles and so the minimal cycle is only defined for a subset of E. In some graphs there is no unique shortest path between two vertices. In these cases there are multiple minimal cycles defined by an edge between two such vertices.

The minimal restriction on the cycle guarantees that the vertices in the path, other than the first and last, are a set. In the presence of a pair of repeated vertices in the path, the pair and all vertices between them, can be replaced by a single instance of the vertex. This will produce a shorter path and thus the original repeated path cannot be minimal. Hence we shall refer to the minimal cycle directly as a set of vertices. The set of minimal cycles in a graph will be explicitly computed by the loop detection algorithm, but in defining our terminology we shall presume the existence of this set, referred to as $\mathcal{M}$.

In order to simplify the presentation of relations that are defined over this set we shall assume that an extra member exists in $\mathcal{M}$. The set of all edges that are not members of a minimal cycle is denoted by the symbol -1. The nominated element completes the set so that the relations that we present are closed over $\mathcal{M}$.

It is clear that each loop in the graph must contain at least one minimal cycle. However there is not a one-to-one correspondence between the minimal cycles on a control flow graph and the loops contained in the program. In particular consider the example shown in Figures 4.2c. This example is indicative of subgraphs commonly found in structured programs. There is a single loop in the graph. Within the body of the loop there is a conditional statement which splits the control flow into two possible paths. Both paths $[c, d, f]$ and $[c, e, f]$ lie on unique minimal



(a) Depth First Spanning Tree

(b) Control Flow Graph With Classified Edges

(c) Two Cycles With A Unique Vertex And Shared Body

Figure 4.8: A Loop With An Internal Condition

cycles through the edge $f \Rightarrow b$. We will return to this example later, showing exactly how we identify the minimal cycles on this graph.

While it is true that some minimal cycles should be merged into a single loop, it is obvious that not *all* cycles on a graph should be merged into a single loop. For any loop detection algorithm based on minimal cycles, the important question that must be answered is:

> *When are cycles independent loops, and when are they separate paths within a single loop?*

One important criteria for merging cycles into a loop is whether or not there is an intersection between the cycles. Any cycle that does not intersect another cycle will be an independent loop. In the case when two cycles intersect they have a shared set of vertices. There must be edges from the unique part of each cycle into the shared set, and also from the shared set into the separate parts of each cycle. The presence of these edges mean that either loop can pass control to the other. This symmetry between the control flows of the intersecting cycles is transitive; if there is an intersection between cycles A and B, and also between cycles B and C then A and C have a symmetry in their control relationships. We define a binary relation on cycles that identifies this symmetry in control-flow:

Definition 4.1. *The binary relation* ***touches*** *(denoted $\between$) over cycles is the transitive closure of intersection on the sets of vertices within those cycles*

1. $A \cap B \neq \oslash \rightarrow A \between B$
2. $A \between B \wedge B \between C \rightarrow A \between C$

Each strongly connected region has a control relationship with the rest of the graph that is defined by two sets termed the *control sets* of the region. The set of vertices that pass control to a region, from outside that region, is called the *entry set*. The elements of the set are referred to as the entries of the region. Conversely, the set of vertices that the region can pass control to, again located outside the region, is called the *exit set*. Each element within the set is known as an exit of the region.

This terminology is different from the definitions common in the literature. The entrances and exits of a region are always defined by the edges that connect vertices inside and outside of the region. Consider an edge $a \Rightarrow b$ where a is located outside a region, and b is located within the region. Normal convention would describe b as the entry of the region. For an edge leaving the region the interior vertex would be known as the exit. We have chosen to redefine these terms for clarity, as our technique operates on a region of interest by tracking the regions that control is passed to and from. Using the sets of exterior vertices contrasts with the normal usage of identifying the vertices within a region that can pass control flow to, and receive control flow from, the rest of the graph.

The definition of touching has a direct case, and a transitive case. When two cycles are touching transitively there must exist at least one chain of intermediate cycles between them. The touching sets operator produces the set of cycles that are present in chains between the two touching cycles.

Definition 4.2. *The* ***touching set*** *of two cycles (denoted $\overline{\between}$) is the set of cycles that form a chain of direct touching between the two endpoints A and B.*

$A \cap C_0$, $C_0 \cap C_1$ $\ldots$ $C_{n-1} \cap C_n$, $C_n \cap B$
$\rightarrow i = 0..n \;\; C_i \in A \,\overline{\between}\, B$

When the strongly connected region under consideration is a minimal cycle X within the control flow graph, we refer to the entry set as X_{IN} and the exit set as X_{OUT}. These sets contain the minimal cycles that control can pass from, and to, respectively.

Definition 4.3. *The* **control sets** *of a minimal cycle X are the sets of cycles with an edge between the cycle and X.*
$X_{IN} = \{C \mid \; C \in \mathcal{M}, \;\; \exists(a \Rightarrow b), \;\; a \notin X, \;\; b \in X, \;\; a \in C\}$
$X_{OUT} = \{C \mid \; C \in \mathcal{M}, \;\; \exists(a \Rightarrow b), \;\; a \in X, \;\; b \notin X, \;\; b \in C\}$

When there is a set of cycles that $touch$ in a flow graph, the control sets are used to determine if the set can be partitioned. Either the set can be split into distinguishable subsets, or all of the cycles are equivalent in control flow and should be merged. Given a cycle X there is a control relation between every cycle in X_{IN} and X. There is also a relation between X and every cycle in X_{OUT}. These relations are used to distinguish the cycles in a set. If every cycle in a set can $touch$ every other cycle in the set then control can pass freely from any loop to any other. In order to distinguish the cycles there must be some difference in the control relations that are external to the set — to the rest of the graph.

To be distinguished a cycle has to possess a control relation that is unique within the set, if cycle X possesses relations with the cycles $X_{IN} \cup X_{OUT}$ then for one of those relations to be unique it must be in the control sets of no other Y in the set — $\forall Y \;:\; (X_{IN} \cup X_{OUT}) \setminus (Y_{IN} \cup Y_{OUT}) \neq \oslash$.

Definition 4.4. *The* **weak control relation** *holds between a cycle X and any cycle Y where there is an input control relation, or an output control relation, but not both, between X and Y.*
$X \smile_w Y \leftrightarrow Y \in (X_{IN} \cup X_{OUT}) \setminus (X_{IN} \cap X_{OUT})$

For our purposes of loop identification there is a symmetry between control flowing into a region, and control flowing out of the region. Given a set of touching cycles it is possible to distinguish the cycle closest to the 'outside' of the set either through a unique control entry, or a unique control exit. There is a second type of distinction that can be made between the cycles in a set. A cycle which has both an entry relation *and* and exit relation with an external region is distinct from a cycle with only one of the relations. A cycle X is also distinct if $\forall Y \ : \ (X_{IN} \cap X_{OUT}) \setminus (Y_{IN} \cap Y_{OUT}) \neq \oslash$.

Definition 4.5. *The* **strong control relation** *holds between a cycle X and any cycle Y where there is both an input control relation and an output control relation between X and Y.*
$X \smile_s Y \leftrightarrow Y \in X_{IN} \cap X_{OUT}$

Both of the control relations in Definitions 4.4 & 4.5 are symmetric. One can prove this symmetry by substitution of the sets in Definition 4.3. The two forms of control relation, weak and strong, are used to define the distinguished members of a set of cycles. Distinguished cycles are those in the set that are differentiated from the rest of the set by a control relation that spans the boundaries of the set. We define two types of distinguished cycles. These two types prioritise the cycles in a set by which way they are distinguished.

Definition 4.6. *A cycle X is* **strongly distinguished** *within a set $\mathcal{L}$ iff there is a strong control relation between X and a cycle Y outside of the set $\mathcal{L}$, and there is no strong control relation between another member of $\mathcal{L}$ and the cycle Y.*
$X \lhd_s \mathcal{L} \ \leftrightarrow \ \exists Y \notin \mathcal{L}\, ,\, \forall Z \in \mathcal{L} \ : \ Z \nsim_s Y \ \wedge \ X \sim_s Y$

Cycles that are strongly distinguished within a set override those that are weakly distinguished. In order for a cycle to be weakly distinguished within a set there must be no strongly distinguished cycles in the set. Cycles that are not control related to a cycle outside the set are not distinguished at all.

Definition 4.7. *A cycle X is* **weakly distinguished** *within a set $\mathcal{L}$ iff there is no strongly distinguished cycle within $\mathcal{L}$, and there is a cycle Y external to the set $\mathcal{L}$ such that X is weakly control related to Y, and no other member of the set $\mathcal{L}$ is weakly control related to Y.*

$$\begin{aligned} X \lhd_w \mathcal{L} \ \ \leftrightarrow \ \ & \nexists Z \lhd_s \mathcal{L} \ \wedge \\ & \exists Y \notin \mathcal{L}\, ,\, \forall Z \in \mathcal{L} \ : \ Z \nsim_w Y \ \wedge \ X \sim_w Y \end{aligned}$$

Given the two ways that a cycle may be distinguished within a set we can define whether a cycle is distinguished from the set or not.

Definition 4.8. *A cycle is distinguished within a set, iff it is either weakly distinguished, or strongly distinguished within the set.*
$X \trianglelefteq \mathcal{L} \quad \leftrightarrow \quad X \triangleleft_s \mathcal{L} \;\vee\; X \triangleleft_w \mathcal{L}$

The Definitions 4.1- 4.8 are properties of cycles on a control-flow graph. These properties are necessary to express our novel definition of a loop. A loop on a control-flow graph is a set of cycles that have indistinguishable control relations with the rest of the graph. This property cannot be determined from the set of cycles alone; or more precisely from the subgraph that they form. In order to define the control relations with the graph outside of the cycle set a context is required. Given a context, and a set of cycles it is then possible to decide if the set forms a loop.

We define a context as two sets of cycles; $\mathcal{U}$ is the set of unclassified cycles, and $\mathcal{L}$ is the set of classified loops. In the particular context represented by $< \mathcal{U}, \mathcal{L} >$ we shall define loops which are subsets of $\mathcal{U}$, and denote each such loop by $\mathcal{N}$.

Definition 4.9. *In the context $< \mathcal{U}, \mathcal{L} >$ a **loop** is a set of cycles $\mathcal{N}$ such that $\mathcal{N}$ is a subset of the currently unclassified cycles U. No cycle in the loop is distinguished from the others $\nexists C \trianglelefteq \mathcal{N}$. All of the cycles in the set touch $\forall C, D \in \mathcal{N} : C \between D$. The set $\mathcal{N}$ is minimal in the sense that no subset of $\mathcal{N}$ forms a loop. All cycles in the loop are either distinguished within the unclassified set $\forall C \in \mathcal{N}, \nexists D \notin \mathcal{N} : C \between D \wedge D \trianglelefteq \mathcal{U}$, or are in the touching set of two distinguished loops $\forall C, D \in \mathcal{N} : C \overline{\between} D \subseteq \mathcal{N}$.*

The application of the loop definition is iterative, for a particular context $< \mathcal{U}, \mathcal{L} >$, a set of new loops $\mathcal{N}' = \{\mathcal{N}_0, ..., \mathcal{N}_n\}$ is defined. This set can then be used to produce a new context:

$$< \mathcal{U}', \mathcal{L}' > \mapsfrom < \mathcal{U} \setminus \mathcal{N}', \mathcal{L} \cup \mathcal{N}' >$$

In each context produced, a new set of loops is defined. Initially the set of classified loops $\mathcal{L}$ is $\oslash$, and the set of unclassified cycles is the set of all minimal cycles in the control graph $\mathcal{U} \mapsfrom \mathcal{M}$. One property that our definition must possess is that every cycle is classified as a member of a loop, for it to match a programmer's intention. If this does not hold, then there are infinite paths through those cycles that are not classified as looping paths. Furthermore, to meet the requirements of the Timing Analysis in Chapter 3 each cycle should only be a member of a single loop. We now offer a simple proof that the loop definition forms a partition of the set of minimal cycles in the graph. The only assumption that we require for this proof is that all cycles are reachable from the entry point.

Theorem 4.1. *Repeatedly applying Definition 4.9 to* $< \mathcal{M}, \oslash >$ *produces* $< \oslash, \mathcal{N}' >$*, where* $\mathcal{N}'$ *is a partition of* $\mathcal{M}$*.*

Proof. For $< \oslash, \mathcal{N}' >$ to be a partition of $< \mathcal{M}, \oslash >$ two conditions must hold; every $C \in \mathcal{M}$ must be in a loop in $\mathcal{N}'$, and each C must only exist within a single loop.

1. C is not within multiple loops. Firstly we observe in Definition 4.9 that once C is classified as being within a loop it is removed from $\mathcal{U}$. For C to be in multiple loops they must be classified on the same iteration. We also observe that each cycle in a loop touches each other cycle. So if $C \in \mathcal{L}_0 \wedge C \in \mathcal{L}_1$ then $\forall X_0 \in \mathcal{L}_0 : C \between X_0$, and $\forall X_1 \in \mathcal{L}_1 : C \between X_1$. By the transitivity of touches, $X_0 \between X_1$ and therefore $\mathcal{L}_0 = \mathcal{L}_1$. Thus if C is in any two loops $\mathcal{L}_0$ and $\mathcal{L}_1$ then $\mathcal{L}_0 = \mathcal{L}_1$ and C is in one loop.

2. Every C is classified as part of a loop or, equivalently, that the iterative application of Definition 4.9 terminates with an empty set of unclassified loops. We prove this property by induction. Firstly consider the base case where $\mathcal{U}$ contains a single loop. This loop must contain an entry from outside $\mathcal{U}$, either from the outermost non-looping scope, or the initial program entry point. Thus the single cycle is distinguished within the set by Definition 4.8, and the iteration terminates. If we add a cycle to $\mathcal{U}$ where all of the previous elements in $\mathcal{U}$ are classified, then there are two possibilities.

Either the new cycle is connected to a cycle already in $\mathcal{U}$ or it is not. In the case that it is connected then it is either distinguished from those cycles or not. If the cycle is distinguished then it must be added to a loop on some future iteration, as in the worst case it will be the final member of $\mathcal{U}$ to be classified. If the cycle is not distinguished then then it will part of the loop that the cycle it is connected to will be classified as. The final case is that the new cycle is not connected to any existing members of $\mathcal{U}$, which means that it must contain an entry from outside, and thus is the same as the base case. This concludes the inductive cases, and so by induction any set $\mathcal{U}$ that we can construct must be completely classified.

$\square$

In control-flow graphs that represent well-structured programs our definition produces the expected set of loops. The motivation for our definition is the result produced on unstructured programs. It is harder to reason about control flow on graphs where cycles overlap, but do not nest, and where loops may form multiple entry and/or multiple exit regions. We argue that the loops that we define are robustly defined on arbitrary control-flow graphs, and that in the presence of these features the resulting loop set is what a programmer's intuition would suggest.

Our basis for this argument is that a programmer's intuition about where the loops in a graph lie is based on the control relationships between regions. If the iteration of one loop is dependent on the control flow in another then those two regions form separate loops. Our definition is the broadest based on differences in control between distinct cyclic regions on the graph.

One unexpected property of the definition is that loops are not defined uniformly on the control graph. The definition has to be context-sensitive in order to maintain a boundary around the unclassified regions in the program. The distinction between loops is formed by their control relationship with this boundary. This has two important consequences:

- Loops are defined according to the context that they exist within. A particular sub-graph may be a loop when embedded in one graph, but not in another. The local structural properties of the sub-graph do not define the loop, only the relationship with the rest of the graph.

- The definition of a loop is iterative. The only invariant property that can be assumed is that all control-flow graphs have a unique entry point. In order to partition the cycles into loops it is necessary to start at this outer point in the graph, and work inwards towards inner loops. This order must be used as these inner loops are not well-defined in isolation, but only in the context of the classified outer regions. In particular each outer loop was originally a set of distinct cycles, but during classification has been merged into a single loop set. This merging process reveals previously unknown information about the distinctness of connections between unclassified and classified regions.

In previous work the criteria for merging cycles into loops has been based on the properties of vertices within the graph. In some cases regions are computed which have a specific property with regard to some vertex, and this region is detected as a loop. In other cases the criteria for adding a vertex to the set of a loop is based on the properties of the vertex with regards to the rest of the set.

Our work is distinct from these approaches as we use cycles, rather than vertices, as the basic element of a flow-graph. Rather than define properties that partition the vertices, we are partitioning sets of edges in the graph. This distinction allows the use of the relative control relationships between cycles to be the basis for defining loops. The other benefit of the approach is that each cycle belongs to exactly one loop, thus we avoid the thorny problem of how to handle vertices that exist within multiple loops.

Our approach can be compared to the ”outside-in” method proposed by Steensgard [70]. The key to both approaches is the recognition that outer loops must be detected first, as the definition of inner loops is dependent on their context. The approaches differ in their representation of context, and the level of abstraction at which they operate.

4.3 Algorithms For Loop Properties in Control-Flow Graphs

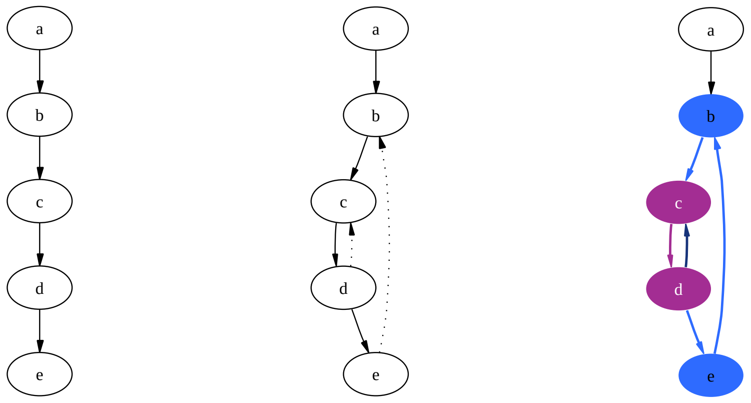
Now that we have a definition of a loop that matches our intuition on arbitrary control-flow graphs we can construct algorithms to detect loop properties. Our goal is to identify enough information to describe the timing properties of a program. In order to do this we need to compute several results from a graph:

1. The minimal cycles on a control-flow graph
2. The loops on a control-flow graph
3. The nesting hierarchy of loops on the graph
4. A partial execution ordering of the loops

The set of minimal cycles is used to define a cutset for the program graph. A cutset on a directed graph is a set of edges that if cut will produce an acyclic graph. This is one of the properties that are required of the loop set L used in Chapter 3. The properties of the edges required for the timing analysis to work are:

- The edge may not be in multiple cycles
- If a pair of loops are ordered (sequencing or nesting) then there must exist a path between the recording points that does not pass through a third recording point.
- Each cycle must contain at least one nominated edge

These properties may be verified from the description of minimal cycles that follows. The first, and last properties are a simple consequence of the edge selected to find each cycle. The second property is more complex to establish, and for now we will simply note that the two orderings required correspond to weakly and strongly related cycles. From this correspondence, and assuming the other two properties, it is simple to show that the nominated edges must be directly reachable without passing the edge of an intermediate cycle. In the next section we shall proceed to construct algorithms that compute each of these results.



(a) Depth First Spanning Tree

(b) Control Flow Graph With Classified Edges

(c) Two Loop-bodies And A Shared Vertex

Figure 4.9: Reducible Loop Nest

4.3.1 Locating Minimal Cycles

The technique to find the minimal cycles is similar to previous work [72, 28, 69]. A *depth first spanning tree* is constructed for the graph. The tree is constructed by traversing vertices of the graph and maintaining a list of previously visited, or *marked* vertices. The traversal starts with the entry vertex of the graph as the current vertex. While there are edges from the current vertex to an unmarked vertex, an edge is selected. The order that the edges are selected in is unimportant, although different orderings will produce a different structure in the tree. The edge is followed, and the target vertex is marked, becoming the new current vertex. The new current vertex is added as a node in the tree which is the child of the previous vertex's node. When the current vertex has no more edges to unmarked vertices, the parent node in the tree becomes the current vertex. The algorithm then continues to look for edges to unmarked vertices, building the tree as they found, and backtracking towards the root node as all of the vertices in a local subgraph are marked. An example of a depth first tree and the graph that it is computed from are shown in Figure 4.9. The example is a well-structured graph with a nesting of natural loops (single entry/exit regions).

The depth first spanning tree has the property that all of the nodes beneath a node in the tree are reachable from the root, through that node. This important property allows the definition of an ancestor and descendant relation on the vertices. As noted above, the structure of the tree is dependent on the edge ordering, and so the ancestor and descendant relations vary with the tree constructed. An *ancestor* of a vertex b is a vertex c where the node for c lies on a path between b and the root in the tree. A *descendant* of a node b is a node c where b is an ancestor of c.

It is now possible to classify the edges in a digraph according to the relation on the tree of the nodes that represent the vertices the edge lies between. Given an edge $b \Rightarrow c$ the edge is classified as a:

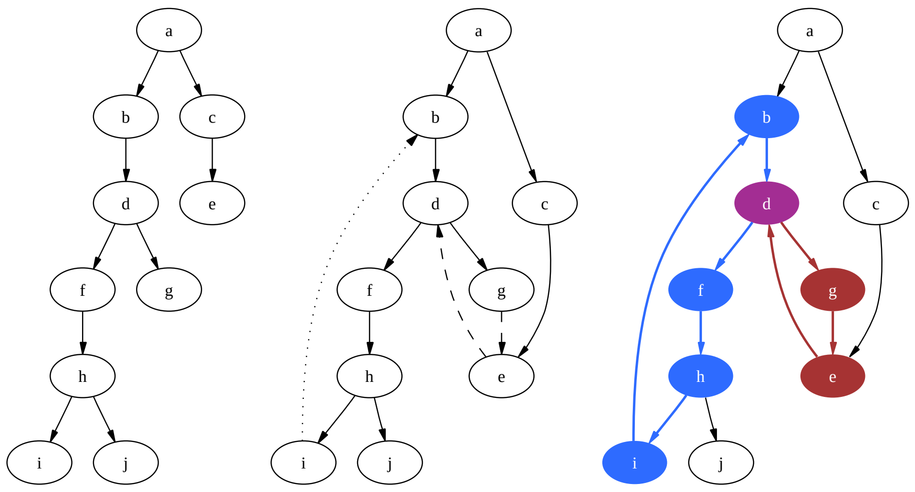
tree edge if the node for c is a child of b

forward edge if the node for c is a descendant of b

backward edge if the node for c is an ancestor of b

cross edge if the node for c is neither an ancestor or a descendant of b

As a tree is acyclic, each cycle on the graph must contain at least one backward or cross edge. To compute the set of minimal cycles we use the simplest approach; each cross or backward edge is used to compute the shortest path from the edge target to the edge source. In the worst case, when the edge does not lie on a cycle, this computation will build a tree containing a node for every reachable vertex in the graph. While there are undoubtedly more efficient methods of computing the set of minimal cycles, this method has the advantage of clarity. It is clear that the set computed matches our definition without a complex proof. Each cycle that is computed is represented as a set of vertices. The total collection of cycles is also represented as a set. This has the benefit that cycles which are permutations of each other are merged into a single cycle. This can be seen on the example graph in Figure 4.10. The smaller cycle of $\{d, g, e\}$ has two possible loop edges as it forms an irreducible regions with two entries. The edge $g \Rightarrow e$ defines the minimal cycle $[e, d, g, e] \rightarrow \{d, e, g\}$. The second edge $e \Rightarrow d$ defines the



(a) Depth First Spanning Tree

(b) Control Flow Graph With Classified Edges

(c) Two Loop-bodies And A Shared Vertex

Figure 4.10: Irreducible Loop Nest

```
$d \leftarrow$ compute depth-first spanning tree from $< V, E >$
**for all** $a \Rightarrow b \in d$ such that edge-type is cross or back **do**
    add shortest path $b$ to $a$ into $\mathcal{M}$
**end for**
```

Figure 4.11: Computing Minimal Cycles on the Graph $< V, E >$

minimal cycle $[d, g, e, d] \to \{d, e, g\}$. As $\{d, e, g\} = \{d, e, g\}$ a single minimal cycle is computed as the two paths are permutations of each other.

The algorithm to compute the minimal cycles is shown in Figure 4.11. For brevity we have omitted the well-known parts; the depth-first tree algorithm and the shortest path algorithm.

4.3.2 Loop Detection

Figure 4.12 shows the algorithm that directly applies the properties of loops from Definition 4.9. The minimal cycles are computed to create the initial context for the algorithm. In this state all of the cycles are unclassified, and held in the set

```
loops =
1: $\mathcal{U} \leftarrow$ minimalCycles(G)
2: $\mathcal{L} \leftarrow \oslash$
3: while $\mathcal{U} \neq \oslash$ do
4:     $\mathcal{D} \leftarrow$ distinguished($\mathcal{U}$)
5:     for all $C_1, C_2 \in \mathcal{D} : C_1 \between C_2$ do
6:         merge $C_2$ into $C_1$ in $\mathcal{D}$
7:         $\mathcal{U} \leftarrow \mathcal{U} \setminus \{C_1, C_2\}$
8:         for all $\mathcal{C} = C_1 \overline{\between} C_2$ do
9:             merge all $\mathcal{C}$ into $C_1$
10:        end for
11:    end for
12:    $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{D}$
13: end while
```

```
distinguished($\mathcal{U}$) =
1: $\mathcal{S} \leftarrow \mathcal{W} \leftarrow \oslash$
2: for all $C \in \mathcal{U}$ do
3:     if $C \lhd_s \mathcal{U}$ then
4:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{C\}$
5:     end if
6:     if $C \lhd_w \mathcal{U}$ then
7:         $\mathcal{W} \leftarrow \mathcal{W} \cup \{C\}$
8:     end if
9: end for
10: if $\mathcal{S} \neq \oslash$ then
11:     $\mathcal{D} \leftarrow \mathcal{S}$
12: else
13:     $\mathcal{D} \leftarrow \mathcal{W}$
14: end if
15: return $\mathcal{D}$
```

Figure 4.12: Loop identification algorithm

$\mathcal{U}$. Each iteration of the outer loop selects the cycles that are distinguished in the set. Definition 4.8 and its implementation ensure that all selected cycles are of the same type; either strongly distinguished or weakly distinguished.

Each distinguished cycle is potentially a separate loop that has a control relation crossing the boundary of the unclassified set. If cycles are touching then they have a symmetric control relation between them. The transitivity of the touching relation means that this control relation may span several intermediate cycles that are not distinguished from the unclassified set. For every pair of distinguished cycles that touch, their touching set is computed. This set contains the intermediate cycles that are spanned by the touching relation. All cycles within the touching set are merged together in the selection set $\mathcal{D}$.

The merged cycles are added to the classified loops $\mathcal{L}$, whilst the unmerged elements are removed from the unclassified set. The outer loop continues until all cycles have been classified. This termination property is guaranteed by Theorem 4.1

The result of the algorithm is shown in Figure 4.10c. The two cycles are correctly identified as independent loops, as the cycle $\{b, d, f, h, i\}$ has a control relationship with the graph exterior to the cycles that the cycle $\{d, g, e\}$ does not. The intuition behind this separation is that iterations of cycle $\{d, g, e\}$ are dependent on the other cycle. If the sink vertex j is removed from the graph, then the control relationships of the two cycles are identical. In this case the algorithm correctly merges both cycles into a single loop.

The irreducible loop nest shown is difficult to analyse as the two loops are not well-structured; they intersect and yet neither is a proper subset of the other. Previous loop detection algorithms would analyse this nest as a single unstructured region. Our algorithm yields a more precise analysis of this type of graph. The result is important as the output from optimising compilers, and low-level code written directly by programmers, features control-flow with these characteristics. Consider the following loop nest in a high-level language:

```
do {
  while( condA ) {
   body...
  }
}
while (condB )
```

The conditions are evaluated at different points in each loop, one prior to the body, and one after the body. There are no instructions between the entry to the outer loop and the entry to the inner loop. So both loops will share the same header at the same location.

4.3.3 Computing The Nesting Hierarchy

For every pair of loops, A and B, in a *well-structured* control-flow graph exactly one of three conditions hold:

- The vertices in loop A are a proper subset of the vertices in loop B.
- The vertices in loop B are a proper subset of the vertices in loop A.
- The vertices of loop A and loop B are disjoint.

Nesting is then defined as a partial order over the loops. When the vertices of one loop are a proper subset of another loop, the smaller loop is *inside* the larger loop. Alternatively the smaller loop is said to be an *inner* loop of the larger. Once control has exited an inner loop it must iterate an outer loop before it can re-enter.

When *arbitrary* control-flow graphs are considered, the condition shown in Figure 4.10 can also occur:

- The vertices of loop A intersect the vertices of loop B, but neither is a subset of the other.

Constructing the nesting hierarchy on arbitrary control-flow graphs requires a novel definition of nesting. It is desirable that this new definition should match the reducible definition on well-structured control-flow graphs. On graphs that are not well-structured it is desirable to preserve the intuitive meaning of a nesting. In the reducible definition the outer loop controls the repetition of the inner loop. Once control flow has exited the inner loop, it must iterate the outer loop in order to re-enter the inner loop.

This intuitive idea is not precise enough to characterise the nesting on loops. Consider an outer loop A with two inner loops B and C. If the inner loops are executed in a sequence, then an execution of loop C must lie inbetween the blocks of iterations of loop A. This definition would characterise sequenced loops as nesting, which is clearly not satisfactory.

From our definition of a loop, every exit from loop B must enter loop A — as loop A is defined as the set of all edges that form cycles through the back-edges of A. Furthermore, every entry to loop B must be from an edge that is present in loop A. In our terminology this requires the two loops to be strongly related.

Furthermore, every such pair of strongly related loops must be nested. Because the loops are independent, they were not merged during loop identification. As the loops are strongly related, they must touch. Therefore they were not

```
tree($O$,$\mathcal{L}$) =
 1: if $\mathcal{L} = \oslash$ then
 2:     return $(O, \oslash)$
 3: end if
 4: $\mathcal{C} \leftarrow \forall L \in \mathcal{L} : L \sim_s O$
 5: $\mathcal{L}' \leftarrow \mathcal{L} \setminus \mathcal{C}$
 6: for all $C \in \mathcal{C}$ do
 7:     $\mathcal{S} \leftarrow$ tree($C$,$\mathcal{L}'$)
 8:     $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{S}\}$
 9: end for
10: return $(O, \mathcal{T})$
```

Figure 4.13: Algorithm To Construct Nesting Hierarchy

merged because there was a difference in their control relations with the loops that were closer to the outermost scope. The loop that is closer to the outermost scope must be the outer loop in the nesting, as all control-flow to, and from the inner loop passes through it.

Given that every strongly related pair of loops forms an edge in the nesting forest, and that the ordering is defined by distance from the outermost scope, we can construct the nesting forest using a depth-first search. An example implementation is shown in Figure 4.13. The initial call uses the outermost scope and the computed loop set; tree($\top$,$\mathcal{L}$).

4.3.4 Computing The Execution Ordering

The possible execution times for an instruction are dependent on the behaviour of any loop nest that the instruction is within. This behaviour controls the periodicity of the instruction, and is used to compute the possible times of execution relative to the loop nest. In order to convert this relative time to an absolute time from the beginning of execution it is necessary to include the contribution of loops that executed before the loop nest was entered.

```
ordering($\mathcal{FP}$,$\mathcal{T}$) =
 1: for all $(L, ?) \in \mathcal{T}$ do
 2:     level ← level $\cup\{L\}$
 3: end for
 4: for all $L_1, L_2 \in$ level do
 5:     if $(L_1, L_2, ?) \in \mathcal{FP}$ then
 6:         before ← before $\cup(L_1, L_2)$
 7:     end if
 8: end for
 9: for all $(?, \mathcal{S}) \in \mathcal{T}$ do
10:     before ← before $\cup$ ordering($\mathcal{FP}$,$\mathcal{S}$)
11: end for
12: return before
```

Figure 4.14: Algorithm To Compute Execution Ordering

The relative ordering of loops during the program execution is a partial order over the loops. The ordering is partial for two reasons; there is no well defined ordering between loops on different levels of the nesting hierarchy, and conditions may make the execution of loops mutually exclusive.

Finding this partial ordering requires tracing possible paths of execution and determining the order that loops occur in it. To simplify this task we use the timing fixpoint computed in Chapter 3. The important property of this fixpoint is that each timing state is computed relative to the last recording point passed. If a timing state exists which records the time from loop A to loop B within the fixpoint then there is a path through the control-flow graph from A to B which does not pass through the recording point of any other loop.

If a pair of loops A and B are strongly related then there will exist a pair of timing states, such that the time $A \Rightarrow B$ and the time $B \Rightarrow A$ exist. However in the case that only one timing state exists a weaker condition holds; the two loops may not be weakly related, but execution of one will follow the other. This is the information that we require to compute the partial ordering.

Figure 4.14 shows an algorithm to compute the partial ordering from the timing fixpoint. For each level in the nesting forest, the fixpoint is examined for timings between sibling loops. It has been established during loop detection, and constructing the nesting forest that the loops are not strongly related. If a timing exists between siblings on the same level of the nesting forest, then it must indicate an ordering in the execution of those loops.

4.4 Timing Equations

The algorithms in the previous section compute properties of the control-flow graph in order to generate a model of the timing behaviour of the program. It is desirable for this model to be the simplest model that describes the timing to a precision that makes the model useful. As discussed at the beginning of this chapter, the main role for the model is to eliminate infeasible paths through the program.

We introduce the following terminology to simplify the results from previous sections:

- I_k is the set of iterations of the loop k. For simplicity all loops will be referred to by integer labels. One such labelling would be the index of a back-edge of the loop in the graph. Computing I_k is beyond the scope of this work, and our system accepts user annotations to specify the iterations of each loop. One system that could compute these annotations is CiaoPP [32]. We assume that given a bound B_k on loop k the set $I_k = \{x \mid 0 \leq x \leq B_K\}$.

- P_k is the set of periods of the loop k. This is the set of times in the fixpoint between each back-edge of the loop.

- $o_k(l)$ is the set of times from back-edges of the loop k to the location l.

Given this set of information we desire a simple expression of the possible times that the location can be executed in. The simplest such expression assumes that the iteration space of the loop is constant, and hence can be described as a linear term. For example a location l within a single loop k is described by:

$$I_k \cdot P_k + o_k(l) \tag{4.4}$$

There are two problems with the formulation of Equation 4.4. The terms are not scalar values, but rather sets of integers. It is not clear what operation is performed by addition or multiplication in this context. While this expression is simple for a program with a single loop, it is not clear how to incorporate the information from an arbitrary control-flow graph. The remainder of this section is devoted to solving these two problems.

Arithmetic Operations The sets of values that will be manipulated are derived from the timing fixpoint. Each set of values contains the possible lengths of execution in paths between two locations. When a pair of terms are added, the underlying operation is to compute the possible lengths of combined paths between points. This addition is a combinatorial operation. If term A contains the execution times between l_0 and l_1, then addition with a term B is only defined if term B contains the executions times between l_1 and some l_2. The resultant term is then the possible execution lengths between l_0 and l_2.

The resultant set must contain the combination of every timing in A, followed by every timing in B. The operation is defined in Equation 4.5.

$$A + B = \{(a + b)|a \in A, b \in B\} \tag{4.5}$$

Given the addition operation on a pair of values the operation over sets is defined as the operation applied to each pair in the Cartesian cross-product of the two sets. We can see from the definition in Equation 4.5, and the associativity and commutativity of addition over integers that the addition operator is both associative and commutative.

Multiplication is more complex. Consider Equation 4.6 which is an attempt to define an associative and commutative operator using the same method.

$$A \cdot B = \{ab|a \in A, b \in B\} \tag{4.6}$$

part 0 = $\oslash$
part n = $\bigcup_{1..n}^{x}\{\ \{x\}\cup$ part $(n-x)\ \}$

Figure 4.15: Computing the partitions of an integer

The operator in Equation 4.6 is associative and commutative, but does not relate to the underlying system. When we attempt to multiply an iteration set against a period set, the result set contains all products between the two sets. But this result does not correspond to the timings of a repeated execution of a loop. If the period set contains two values p_0 and p_1, and the iteration set contains $0 \dots 4$, then the operator in Equation 4.6 produces the powers of p_0 and p_1 up to the fifth power.

The resultant set of timings is equivalent to a *constant* choice of path through the loop to be executed on each iteration. The desired behaviour is that on each iteration, all possible paths are chosen, and the result set contains all combinations of the paths possible in the number of iterations. We can observe immediately that the operands for the desired operator will be treated differently, and hence the desired operator will not commute.

The left operand will specify the number of iterations. The partitions of an integer are sets whose sum is the integer. For example, the partitions of the integer 4 are the elements of the set $\{\{4\},\{3,1\},\{2,2\},\{2,1,1\},\{1,1,1,1\}$. These partitions are the selections of how many times each path is executed in a loop with the total number of iterations. A recursive definition of the partitions of an integer is shown in Figure 4.15.

The right operand specifies a period. The partitions of each iteration select how many times a period should be repeated. Each element in the partition needs to be combined with each possible period. The elements of the partition define the weight of each period in a particular combination of paths through the loop. Equation 4.7 shows how the elements of the partitions and the individual periods are combined in the multiplication operation.

$$\begin{aligned} I_k \cdot P_k = \{n \cdot t \ &: \ i \in I_k,\ n \in N \in (\text{part}\, i) \text{ s.t. } |N| \le |P_k|, \\ &\quad t \in T \subseteq P_k \text{ s.t. } |T| = |N|\} \end{aligned} \tag{4.7}$$

Generating Equations From Program Structure In order to express program structure as equations we need to introduce some more terminology for the structural results in the previous section:

- $l \subset L_k$ is true if the location l is inside the loop L_k, or if L_k is a ancestor of a loop containing l in the nesting hierarchy.
- $L_K < l$ is true if the loop L_k is before the location l in the ordering computed.
- $t(l)$ is the set of times on loop-free paths from the program entry point to the location l. These are the timings in the fixpoint for the location with a recording point of -1.

The expression for the timing set of a location in an arbitrary control-flow graph is given in Equation 4.8. All of the required values, apart from the iteration space of the loops, is provided by the algorithms in the previous section. Without the iteration spaces the equation is a generating expression that maps a set of iteration spaces onto timing states. If the iteration spaces are substituted into the equation then a concrete set of values is produced.

$$\sum_{l \subset L_k}^{k} I_k \cdot P_k \; + \; t(l) \; + \sum_{L_i < l L_i}^{i} \{|I_i|\} \cdot P_i \tag{4.8}$$

4.5 Example Timing Schema

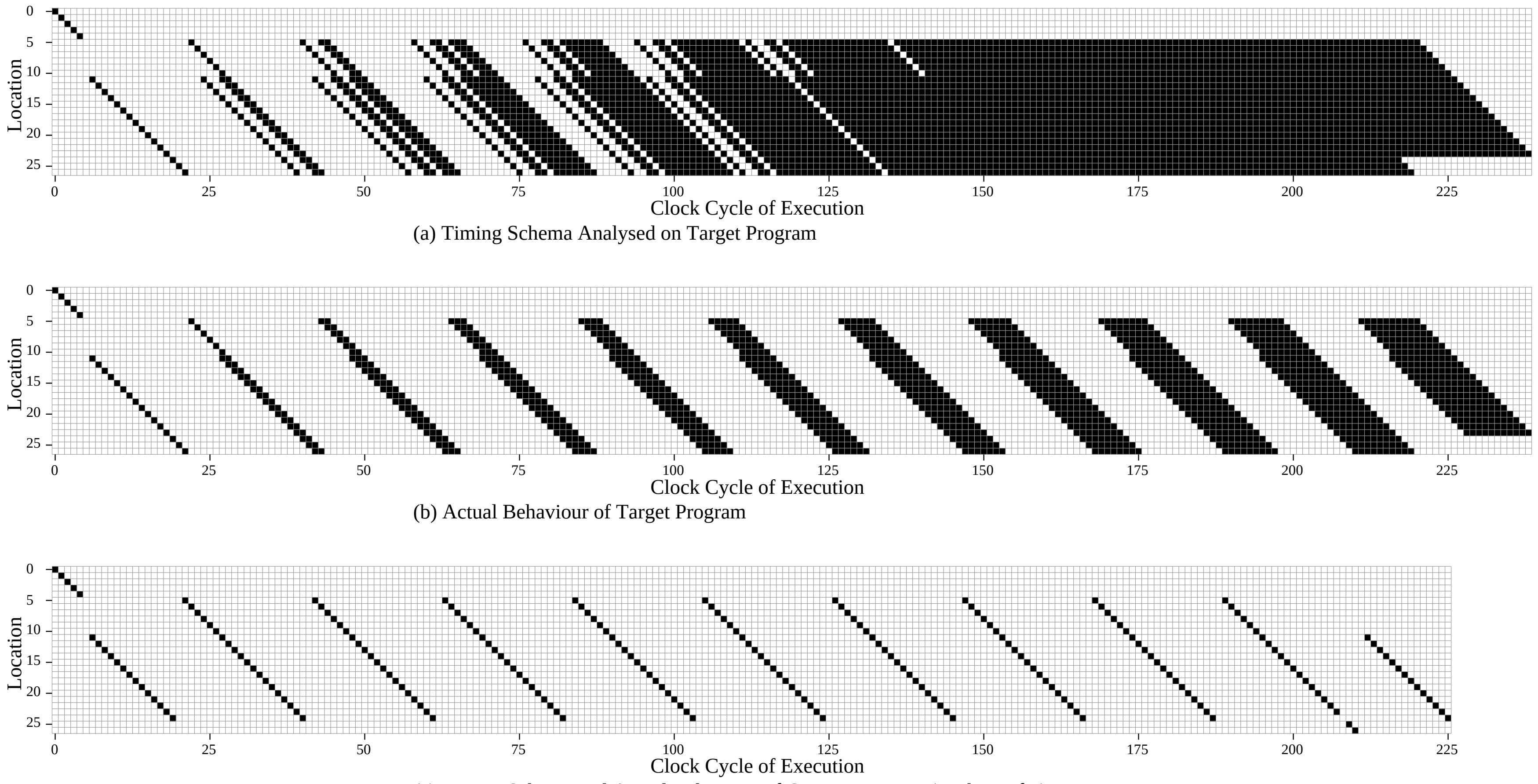
To conclude our discussion of Timing Analysis we apply the techniques in Chapter 3 and this chapter to a sample program. For completeness we will use the example program in Chapter 3. This provides a simple example, and although it does not contain any nested loops the techniques described are applicable to more complex program structures.

The Timing Analysis fixpoint in Chapter 3 has been used to determine loop periods and the timing offsets of each location within each loop. The loop detection has been applied to identify the loops in the program. The particular structure of the program includes three minimal cycles which are correctly identified as a single loop.

Identifying the bounds of loops is outside of the scope of this work, and so annotations have been supplied that provide the bounds of the loop. This information is sufficient to resolve the timing equations to concrete terms and the results are plotted in Figure 4.16a. For each row in the graph, the clock cycles in which that instruction can execute are represented by the filled boxes. The program bugs identified in Chapter 3 cause the timing to diverge on each iteration of the loop, until the timing is saturated. Towards the end of the program execution it is not possible to determine which instructions are executing at which times.

The result shown is an over-approximation of the timing behaviour of the program. In particular, one of the three cycles is actually an exit path for the loop. This cycle is only executed on the final loop iteration, and not on previous iterations. It is not possible to determine this fact based on the static control-flow of the program. Analysis of the program variables, and their effects on conditions in each iteration is orthogonal to the research presented within this thesis. To illustrate the loss of precision that this missing information causes the diagram in Figure 4.16b shows the actual timing behaviour of the program.

Finally, Figure 4.16c shows the ideal behaviour of the program. When the bugs in the program are corrected this is the expected timing behaviour. We show all three cases to make the point that while the loss of precision between the first two cases is noticeable, it is small in comparison to the difference between either case and the ideal behaviour. This makes the Timing Analysis suitable for a designer to verify that a program is operating as expected by examination of the timing model produced. Timing models which saturate, such as Figure 4.16a, are clearly dintinguishable from models in which the times of possible execution have not saturated while the program is running.



(a) Timing Schema Analysed on Target Program

(b) Actual Behaviour of Target Program

(c) Timing Schema and Actual Behaviour of Correct Program (With Bugfix)

Figure 4.16: Timing Schema of Target Program

Part II

Precision Semantics

Chapter 5

The Multi-Precision Domain

When arithmetic operations are implemented in computers a binary representation is normally used, as mentioned in Chapter 2. There are two common formats for these strings of binary digits. In a format where the value of each bit is fixed, the radix point is always between the same two positions in the bit-string. The static position of the radix point gives rise to the name *fixed-point* format. This format has the advantage that an implementation in digital logic is simple. The other common format allows the radix point to move within the binary string. This *floating-point* format is more expensive to implement as the position of the radix point is now a state that must be maintained.

There is a trade-off between the complexities of the two formats. While fixed-point format is the less expensive format in terms of size and power consumption it is more complex for the programmer to use. Because the position of the radix point is implied, rather than explicitly specified, the programmer must track it manually. Each operation modifies the format as the value of each bit position changes. Although the fixed-point format provides more efficient execution, it is at the expense of a more complex programming model.

When a programmer manipulates floating-point values it should be safe to assume the values are simply numbers; and behave as well defined mathematical objects. The simple arithmetic operations have the same associativity and commutativity as their counterpart operations on rational numbers. The similarity between the mental model of the programmer of what they are trying to achieve and the expression in the programming model makes the language expressive.

The properties of the floating-point format that make it desirable as a programming model only exist when the length of the bit-string is dynamic. In this *true* floating-point format the ability to expand the bit-string is necessary to preserve the relation to rational numbers. When the representation is restricted to a constant length the objects represented by the binary strings cease to behave as numbers. This complication arises in practice because the native number formats used in hardware are a fixed *length* bit-string.

A simple numerical model is presented to the programmer as a result of hiding representation details. The floating-point format hides the radix state from the programmer making a simpler model than the fixed point format. This simplicity is lost in practice as restricting either format to a constant length of string causes a loss of precision. For the arithmetic operations the range of the input is smaller than the range over the combination of input and output. In order to store the result of an operation in the same fixed size as the input some form of truncation is necessary.

The truncation of floating-point operations on real hardware is invisible from the programmer. Although it is specified exactly, this specification is separate from the languages that provide floating point operations. The values that are manipulated by programmers of such languages do not possess the properties of numbers, which makes it difficult to reason about their numerical accuracy and behaviour. Our hypothesis is that a language that operates upon arbitrary precision floating-point numbers, but generates fixed-point code for execution, will result in a more expressive programming environment and more efficient code.

To test this hypothesis we have designed a language for writing reactive filters. Each filter is a reactive process; a set of equations that are evaluated in a loop. For each iteration of the loop a set of input values are received and computed upon to produce output values. The input and output values are communicated with the environment on each iteration. A set of stateful variables retain values from one iteration to the next. The remaining variables in the equations are transient being recomputed in each iteration.

Each filter process is an infinite loop that processes a stream of input values, and retains some state over the loop, to produce a stream of output values. The precision of each of the input, output, and stateful variables is provided in the variable declaration. This precision declaration defines the length of the bit-string, and the value of each position within the string. We apply a static analysis that determines the minimal precision required by each transient variable. The semantics of the operators within the language are defined as multi-precision arithmetic. The complete set of precision declarations parameterises the multi-precision arithmetic to a specific set of operations. A compiler is *automatically generated* for the specific set of operations in each program from an interpreter of the multi-precision operations.

5.1 Reactive Process Language

The goal of a *domain-specific-language* (DSL) is to use knowledge about a specific problem to reduce the complexity of programs specific to that problem domain. Programs that solve similar problems can be expressed with a similar structure. This common structure can be removed from the programs and encoded in the language. The result is that the DSL is more expressive as programs have a simpler structure and more clarity.

Each of the programs in our reactive process language is expressed as a set of equations. The single infinite loop is implicit, being defined in the semantics of the language. The program uses a single assignment form, each equation has a single variable on the left hand side being assigned the evaluation of the right hand side. Each left hand side in the program is a unique variable. The restriction that each assignment can only be made once is not sufficient to ensure a single consistent solution for the equations. Consider the simple program:

```
loop {
    x = y+1
    y = x+1
}
```

There are two possible evaluation orders for the pair of equations, which both yield different answers. If we assume that the state of the program is $\{x \mapsto 0, y \mapsto 0\}$ then the state after an iteration could be either $\{x \mapsto 2, y \mapsto 1\}$ or $\{x \mapsto 1, y \mapsto 2\}$. To guarantee that our language semantics provide a *unique* mapping of the state across iterations we must provide some ordering constraints as follows. The variables in the language are divided into four types:

Input variables are never assigned by the programmer. Each variable is associated with an external stream of values. On each cycle of the process the head of the stream is assigned to the variable, and the stream is reduced by one value.

Output variables are also associated with a stream. Each cycle the value assigned to the variable is appended to the stream.

State variables are persistent; their assigned value is retained from one cycle of the loop to the next. The value assigned in one cycle will be the value used on the right hand side of equations in the *next* cycle.

Temporary variables are transient. The values of Temporary Variables do not persist across loop iterations. Their usage implies an ordering as they must be defined on the left hand side of an equation before they can be used on the right hand side.

The types prevent cycles in the dependencies between program variables. In the example above the assignment to x depends on the value of y, and the assignment to y depends on the value of x. Any cycle in the variable dependencies will allow different orderings of the evaluations as there is no well defined starting point. The input and output variables are explicit starting and ending points respectively. There are no dependencies in the evaluation of an input variable, and no variable is dependent upon the value of an output variable.

The state variables in a program are essential to breaking cycles in the dependency order. Regardless of the order of declarations, each usage of a state variable is sequenced *before* the definition of the variable. Thus the value in a state variable that is used during a loop iteration is the value computed in the previous iteration. For this reason each state variable *must* be initialised to a value at declaration.

The sequencing constraints on Temporary Variables are the inverse of the State Variable constraints. Each Temporary Variable must be defined before being used. No explicit declaration is required as a value for the variable will be defined in each iteration before it is used in an evaluation.

Two conditions are placed on the program to ensure that it defines a set of equations with a uniquely defined evaluation order. Implementations of the language enforce these conditions, rejecting programs that fail to match them as illegal.

Single Assignment. Each variable in the program must be assigned exactly once in the program.

No Stateless Cycles. There must be no strongly connected components in the dependency graph of temporary variables. Every cycle in the dependency graph must contain at least one state variable. The semantics of state variables are those of simulation variables, holding both a current and a next value. There is no dependency between these values *within a single loop iteration* thus avoiding any dependency cycles and ensuring that there is a stable result to each computation.

Expressing a program as a set of independent equations ensures that the syntax of the DSL is as close to the domain as possible. In Figure 5.1 we present our motivating example within the domain. The Kalman Filter [79] is used heavily in embedded systems, for example GPS [25] receivers and other location systems [62]. The filter is defined as a recursive set of equations, on each iteration of the filter a new measurement z is used, and compared with a prediction z^+. The residual error term is used to update the internal model variable. During operation this model reacts with the environment in two phases. The measurement phase

$$\begin{aligned}
x^- &= a \cdot x \\
p^- &= a^2 \cdot p + q \\
z^+ &= h \cdot x^- \\
r &= z - z^+ \\
k &= p^- \cdot h \cdot (h^2 \cdot p^-)^{-1} \\
x &= x^- + k \cdot r \\
p &= (1 - k \cdot h) \cdot p^-
\end{aligned}$$

Figure 5.1: Definition of a Kalman Filter Over One State Variable

updates the state of the model based on the new input from the environment. In the predication phase a new output is produced based on the current state of the model. The importance of the filter is the use of an internal model to smooth out noise in the samples; this adaptivity underlies its use in positioning systems. The ability to represent such forms of feedback in the sample programs without any cycles in the dependency tree is an important characteristic of our language design.

Figure 5.2 shows a Kalman Filter implemented in our DSL. A single measurement variable, and a single state variable are used. We use a single variable for simplicity, there is no reason the analysis could not be applied to more complex state vectors. The declarations for input, output and state variables are all parameterised by a precision. The precision parameter specifies the length of the bit-string stored in the variable. All of the bit-strings are a constant length. In the case of inputs and outputs these are defined by the environment that the process executes within. The size of the state strings are defined by the designer. There is a tight limit to the amount of storage on the target device and so we leave the decision of how to partition this storage amongst the state variables to the designers. The precision of the temporary variables is implicit in the program.

The variables declared in a reactive process include a precision parameter of

```
input<0:-7>  z;
output<0:-7> zpred;
state<7:-7> k = 1/2;
state<3,-4> h = 1;
state<0,-7> q = 1/64, residual = 0, xbar = 1/64, pbar = 0,
            x = 1/64, r = 1/32;
state<1,0>  a = 1;
state<-4,-6> p = 1/32;

    xbar = a * x;
    pbar = a * a * p + q;
    zpred = h * xbar;
    residual = z - zpred;
    k = pbar * h * inv(h * h * pbar);
    x = xbar + k * residual;
    p = (1 - k * h) * pbar;
```

Figure 5.2: Kalman Filter Implementation

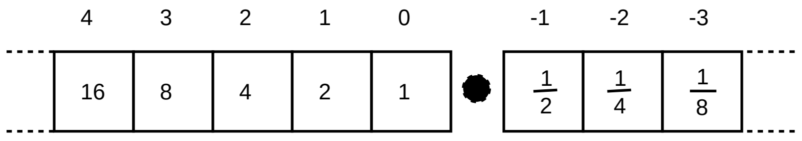


Figure 5.3: Bit-strings over the Binary Numbers

the form `<3:-2>`. This pair of numbers define how the bit-string is mapped onto the bits stored in the variable. In the parameter `<u:l>`, u is the most significant position in the string contained in the variable and l is the least significant position. The form of the bit-strings is shown in Figure 5.3. The positions that are greater than zero contain the integer bits. The binary point is between positions zero and minus one, with the negative positions containing fractional bits. Each position n retains the value 2^n from the definition of a binary integer and n is allowed to range freely over the integers $\mathbb{Z}$.

A *value* in the DSL defines the bits at all positions $n \in \mathbb{Z}$, although only a finite range of these bits are represented. Each value is comprised of a tuple of a finite length bit-string, an uppermost position and a lowest position. Each of the positions is related to the other by the length of the string. Each of the positions within the string has a positive value bar the first. The values are two's-complement notation, with the first position having a negative value and being referred to as the sign of the value. The string is sign-extended; all of the positions above the uppermost are defined to have the same digit and negative value. Operations on values round towards negative infinity; every digit beneath the string is defined to be zero.

An analogous system can be defined for other rounding modes, such as rounding towards the nearest even number. Our choice of rounding mode matches the operation of the PIC architecture, and so leads to an efficient implementation in this case.

The domain of the values defined in the language is the set of infinite binary strings. These strings are expansions of the reals. The values that are instantiated within the program are approximations of these reals. These instantiated values are *finite binary strings* and can be of arbitrary length and position within the infinite strings. Each finite binary string defines the value of every bit in an infinite binary string.

The operations on finite binary strings are defined over the source strings and positions to produce both a resultant string and a resultant position for that string. Operations are defined between all finite binary strings because each such string defines all of the bits in an infinite string. Thus the addition, subtraction and multiplication operators form a field over the set of values. The operators in the language have the same properties as numbers and combinations of them will provide the programmer with the expected results.

Each precision for a variable defines the set of finite strings that can be represented, and thus the set of values the variable can hold. Efficiency of the code generated from the DSL requires a uniform distribution of values over a given range. The uniformity of the distribution of values allows their representation as integers. Each integer is a scaled representation of the underlying value. As integer mathematics is usually more efficient than a floating-point representation this *fixed-point* form results in an efficient implementation. Some fixed-point formats fix the number of bits in each value and where the implied radix point lies between those bits. For expressiveness we want our variables to be of arbitrary size, and for efficiency we want to fix a scaling transformation between the underlying variables and a fixed-point representation. We argue that the language we describe allows the programmer to implement a filter apparently using arbitrary precision arithmetic.

$$x_{u,l} \in \left\{v \cdot 2^l : v \in \{-2^{u-l}\ ,\ 2^{u-l} - 1\}\right\} \tag{5.1}$$

For example the variable $x_{1,-1}$ would represent the values -2, $-\frac{3}{2}$, -1, $-\frac{1}{2}$, 0, $\frac{1}{2}$, 1, $\frac{3}{2}$.

In the final stage of any filter there is an assignment of a value to an output. This assignment may cause a truncation in the value, if the set of bit positions in the value differs from the set of positions that the output variables is declared over. In this situation there is a well-defined truncation of the bit-string. This truncation causes bits to be dropped from the string, and thus there is a loss of accuracy in the truncation.

Truncation is associative but it does not distribute over the operations in the language. If T_0 and T_1 are two particular truncation operations on a binary string, then $T_0(T_1(x)) = T_1(T_0(x))$ for all binary strings x. However, if $\cdot$ is a binary operator over the strings then $T_0(x{\cdot}y) \neq T_0(x){\cdot}T_0(y)$ for any non-trivial truncation T_0. If a truncation occurs before an operation then the lost bits do not affect the end result. This sequence contrasts with truncating a value produced by an operation. In the latter case the bits being removed will have influenced other bit positions in the final value.

The code without the size parameters for variables defines an arbitrary precision implementation of the algorithm. This implementation would be executable if the precision parameters of each variable are maintained dynamically. This execution environment would have to reallocate storage for variables when necessary.

On our target domain this dynamic memory management is infeasible and our interest is finding a static description of the program using the parameterisation that is provided.

This constraint on the size and range of each persistent variable is defined by the available storage on the target device and how the programmer wishes to allocate it to individual pieces of the state. Each *state* variable must be large enough so that it does not cause inaccuracies in computed output and yet small enough so that the state fits into the memory of the device. In writing the definition block the programmer annotates the desired precision of persistent values in the algorithm.

There are two cases for assignment in the language, the precision of the target variable can be:

fixed as it is a *state* or *output* variable. When the value produced by the expression is at a different precision to the target precision we truncate or expand the value as necessary. When truncating we remove excess bits, where the removed bits are in positions above the target bit-string we are storing the value modulo the range of the target. Where the removed bits are in positions below the target bit-string we are rounding towards negative infinity. When expanding the string to fill the target variable we add 0's beneath the string, and sign-extend the string above.

unknown as it is a *temporary* variable. When the target has an unknown precision we perform a *Precision Analysis* to define the precision of the variable so that any results computed from it will be bit-correct.

The semantics of our language define the computed output streams to be the bit-correct result with respect to the persistent values in the filter and the input streams. This definition requires a set of precisions for the *temporary* variables that do not propagate errors from underflow or overflow into the *output* variables. In order to compute this set the *state* variables must be annotated with their desired precision. Without a predetermined solution for the state variables, most programs will tend to increase the precision required on each cycle. This clearly tends to infinite precision and has no conservative approximation that we can use. The state variables are also a natural place to clamp the precision as the designer knows how much memory is available to hold the state between iterations.

5.2 Target

Our target device is a PIC-16F84[47] micro-controller. We have chosen this device as it is representative of many real-world applications in the robotics and pervasive computing fields. The main attractions of the device are low cost (under a dollar) and low power consumption (a few milli-watts). Unfortunately this simple design creates limited resources and an awkward programming model as described in Section 1.2. In addition to these general concerns, we now detail issues that are specific to generative programming for the PIC.

The program and data memories are not shared, and it is not possible for the device to modify its program memory while it is running. This limits the possibilities for generative or self-modifying code and makes the analysis of PIC programs easier.

The instruction set of the PIC supports basic arithmetic operations (addition and subtraction) but more complex operations (eg multiplication and division) must be written by the programmer. There is support for logical AND, and both inclusive and exclusive OR. Control flow is limited, with decisions being performed by conditional *skip* instructions. These can either execute the next instruction, or skip over it, depending on the value of a bit in the register file. Logical shifts are not supported, although rotation can be performed through a register and the carry bit, a single bit at a time. Logical shifts can be constructed from combinations of rotate sequences and masking.

The PIC uses a normal model of control-flow; each instruction in the program has an integer label and a register called the PC selects the current instruction. Execution of each instruction affects the PC and so directs the flow of control through the program. The important point is that the target machine is imperative and the active program point is part of the state being passed through the computation. This differs from the control flow in our meta-language and we shall explain how we have embedded this imperative state machine in a logic language in Section 7.3.

The code density of PIC programs is low because of the decisions made to select this instruction set. Logical shifts and rotations of multiple bits take several instructions which are costly both in processor cycles, and in slots in the limited program space. Each conditional split in the control flow must be constructed from multiple instructions. Implementing multiple precision arithmetic is costly as we must extract bit-strings from different locations and merge them in order to perform operations. This requires both shifting of data within registers and conditional code to interpret values at different precisions. However, as we will show this implementation can be achieved by specialising away the conditional code flows and choosing efficient combinations of operations. Implementing arbitrary precision (that is dynamically changing precision) would not be feasible because this approach could not reduce the code synthesised for the PIC.

5.3 Compilation Process

Our technique compiles the DSL program into a program binary for the target device. This process is split into two phases:

Precision Analysis infers the precision parameters for each temporary variable. The analysis is static, operating entirely during compilation and computes a safe approximation of the precision necessary to compute values according to the multi-precision semantics. The novelty of the technique is the transformation of the precision problem into a logic language CLP(FD) and the use of an existing CLP solver. The technique is more precise that previous work in the field. We describe the Precision Analysis in Chapter 6.

Efficient DSL Compilation is a meta-programming technique that generates an efficient version of a parameterised language. An interpreter is used to execute the DSL language parameterised to a specific precision. Specialisation is used to automatically construct a compiler from the interpreter. The technique is described in Chapter 7.

Chapter 6

Analysis of Variable Precision

Our reactive process domain specific language (DSL) is described in Chapter 5. Each variable in a program is declared as being one of four types. Of these types all but the transient variable declarations include a precision parameter. This parameterisation defines the number of bits that the variable holds, and how the bit positions in the variable map onto the bit-strings used in the arithmetic model.

The *Precision Analysis* uses these declared precisions to infer a set of precisions for the transient variables in the program. The semantics of the DSL define arithmetic that operates over multi-precision variables. Approximating this arithmetic by operations on fixed size variables implies that truncation must occur when the bit-string is stored. Each of the arithmetic operators may produce a result that requires more bits to represent than the source bit strings. The values that are computed in the program tend to increase in size as operators are applied to them. Therefore some truncation of values is inevitable in most programs.

The goal of Precision Analysis is to infer a set of precisions for the unknown variables that compute the same results as the original arithmetic on arbitrary sized strings. In order to infer this set correctly the analysis must choose precision parameters so that the result of truncatation multiple times throughout the computation is identical to the single truncation defined in the original program.

As discussed in Chapter 5, a program variable X in the program is represented as a tuple of three values. In this Chapter we shall use the notation $X_{u,l} = S$. The parameter u is the index of the most significant bit in an infinite bit-string. The value of the first bit in the string S_0 is -2^u. The values of the remaining bits in the string S_n are 2^{u-n}. The parameter l is the least significant bit in the variable where S is the finite portion of the bit-string such that $u - l + 1 = |S|$.

The precision of each variable is a logarithmic representation of the bounds on each value. These bounds are the upper and lower bounds on the range of the variable and a bound on the minimal difference between representable values. Using a logarithmic representation produces *linear* constraints between the precisions of variables for the set of arithmetic operations.

Restricting the problem to linear constraints between variables allows a simpler solution at the expense of a less precise solution. Consider the addition of 100 single bit unsigned variables, each can take the values $\{0, 1\}$. A logarithmic representation encodes each variable as an upper bit position of 0, and a lower bit position of 0; $X_{0,0}$. When two such variables are added the result can overflow by a single bit; $X_{0,0} + Y_{0,0} = Z_{1,0}$. There are values in the range of Z that cannot be produced by the sum, but this information is lost at each stage of the analysis. The result of adding all 100 variables together is a representation that is 100 bits in size; $R_{100,0}$.

A more direct representation of the precision would be to store the interval that the variable covers. In this representation each variable would be $X_{[1,0]}$ to show that the variable can range between one and zero. Performing the addition on these intervals would yield; $X_{[1,0]} + Y_{[1,0]} = Z_{[2,0]}$. After the same 100 additions the result would be an interval $R_{[100,0]}$. Converting the interval to the logarithmic representation at the end would produce $R_{[100,0]} \mapsto R_{7,0}$ as $100 < 2^7$. This is clearly a tighter bound on the possible values produced, but for computations containing multiplication the constraints between variable precisions are no longer linear. Solving non-linear constraints is much less efficient and more complex than the restriction to logarithmic representations.

Operation	Constraints
$x_{u,l} = a_{au,al} + b_{bu,bl}$	$u \leq \max(au, bu) + 1, l \geq \min(al, bl)$
$x_{u,l} = a_{au,al} - b_{bu,bl}$	$u \leq \max(au, bu) + 1, l \geq \min(al, bl)$
$x_{u,l} = a_{au,al} \cdot b_{bu,bl}$	$u \leq au + bu, l \geq al + bl$
$x_{u,l} = \text{inv}(a_{au,al})$	$u \leq 1 - al, l \leq 0 - au$

Figure 6.1: Forward propagation constraints

6.1 Forward Propagation

The loss of precision that occurs from the analysis of each step of a computation is due to taking a worst case assumption. In the analysis of an operation on two precisions the resultant precision is large enough to hold all possible outputs. However, not all of the possible values in the resultant precision are produced by the operation. The information about which values cannot be produced is lost at each step. The assumption being made is that the largest result of the operation can occur.

Figure 6.1 defines the propagation of precisions through the operators in the language. The parameters for the resultant precision of operator are inequalities on a function of the input parameters. For the first three operations the input parameters and the operator define the largest possible result. The inequality identifies this size as a bound on the precision. There are smaller possible results, some of which may be correct, but there is not enough information in this propagation to decide which can be used.

Inversion is a separate case from the other operators as it is not closed over the finite binary strings. On a scalar value the inversion operator takes the reciprocal of the value. The smallest example of a value whose inversion is not in the set of finite binary strings is the integer 3; $\text{inv}([0, 1, 1]_{2,0})$. The result is the value $\frac{1}{3}$ which has an infinite binary expansion.

The propagation of precision defined in Figure 6.1 operates forward through the program. Once the equations in the program have been ordered into a sequence for evaluation, each equation can then be parsed into a binary tree. Each operator in the expression is a node in the tree. The inversion operator is unary and so will only contain a single child node. The leaves of the tree all represent variables with known precision. The variables could have declared precision as they are state or input variables, or alternatively they could be temp variables. In the latter case the precision of the variable would be known as it will have been defined by a previous expression tree in the sequence of the program.

The forward propagation is used to compute the precision of a node when the precision of both children is already known. When the precision of the operation at the root of the tree has been evaluated it can be stored as the precision of the target variables. In a program containing only addition, subtraction and multiplication this would be sufficient to compute bounds for each variable, although they may not be precise.

For the inversion operator it is not possible to compute a lower bound on the lowest bit position in the string. As each value is a string of binary digits we can split the string into two pieces at the radix point and talk about the string above the point and the string below the point. These two strings of binary bits can both be interpreted as integer values.

In order for a finite binary expansion to exist the denominator of the reciprocal must be either one, or a power of two. This implies that one of the two integer strings must be zero and the other must be a power of two. Any value in which this is not the case will have an inversion with an infinite binary expansion. The bound that we place on the lowest bit of inversion in Figure 6.1 is the position produced when the number has a finite expansion. No bound can store a non-finite expansion without error so an arbitrary choice of bound that can store the finite cases is suitable.

It would appear that an inversion operation will cause a loss of precision in the output of a computation. However, although an infinite expansion is produced, only a finite number of bits of the expansion *affect* the output of the computation. If a precision for the inversion result is chosen that is correct for each bit of the output then the loss in precision cannot be observed by the programmer.

The forward propagation process operates on the binary expression trees described above. Each equation in the program is parsed as a binary tree; each node is an operation. The single outgoing edge from each node to the level above in the tree represents the result. The two incoming edges are the operands. Each leaf in the tree is a variable with a known precision. The result of the root operation of each tree defines a precision for the target variable. All of the other nodes in the tree correspond to temporary results created during the evaluation of the expression. The precision of each implicit temporary variable is stored as it may be refined later in the Precision Analysis.

This process is applied to the trees of expressions that are assigned to variables with an unknown precision, in the order defined by the variable types. For each operation the bounds on the precision of the result are computed according to the constraints in Figure 6.1. The most conservative bound is used in each case, each inequality in the table is considered to be an equality, and the precisions of each temporary variable, both implicit and explicit, is stored as the variable's precision.

6.2 Backward Propagation

The forward propagation process has inferred a precision that is *sufficiently* large to hold each result in the computation. This solution alone is not the most efficient use of memory resources. A second process called Backward Propagation is used to refine this solution by computing which bits are *necessary* in the result. Only those bits that affect the final result in a computation need to be computed.

Normally the final assignment in the filter is a truncation. Each truncation operation loses information. The operation maps many precise values onto a single imprecise value. If bits have only been used in the preceding chain of operations to affect these lost bits then those parts of the computation are redundant. Further truncations can be introduced to the computation earlier in the chain of operations without altering the final result.

Each truncation that can be introduced is equivalent to reducing the size of an intermediate variable — as long as it does not affect another result using the same intermediate variable. The aim in propagating necessary information backwards through the computation is to add these possible truncations as early as possible. Each bit that is removed from a variable improves the tightness of the analysis producing a shorter computation that requires less storage.

Each backward constraint placed on the computation can only eliminate bits that have no effect on the truncated bits of the output. For the addition operator there are two sets of bits that can be eliminated from the computation without producing an effect in the bits of the result.

Bit positions more significant than the result. The value of a bit in an operand cannot affect a less significant bit in the result. Each bit in a string is the coefficient of a power of the radix of the string. Adding a higher power of the radix to a value cannot change the value modulus a lower power of the radix. I.e., the carries in addition only propagate to the left of the string, never to the right.

Consider the operation; $X_{1,0} = [1, 0, x, 0, 1]_{4,0} + [1, 1, y, 1, 0]_{4,0}$ where both x and y have one of the values $\{0, 1\} \cdot 2^2$. No matter what the value of x or y is, the value of positions 0 and 1 in the result will always be $[1, 1]$.

Bits less significant than the bit positions in the operand. Our definition of the results of operators and truncation of values both round towards negative infinity. Each bit position that is less significant than the positions represented in the string is implied to be zero. As adding zero to a bit has no effect there is no reason to add an implied section of a bit-string to a represented section. When one operand in an operation has less significant bits than the other operand those bits can be removed from the computation.

The constraints that we use to propagate precision in the backward direction are shown in Figure 6.2. Addition and Subtraction can remove redundant bits from the computation according to the properties described above. The multiplication operation can also attempt to remove more significant positions from its operands. This is less straightforward than the addition case as the bits in a multiplication

Operation	Constraints
$x_{u,l} = a_{au,al} + b_{bu,bl}$	$au \leq u,\ bu \leq u,\ bl \geq \text{min}(bl, al),\ al \geq \text{min}(bl, al)$
$x_{u,l} = a_{au,al} - b_{bu,bl}$	$au \leq u,\ bu \leq u,\ bl \geq \text{min}(bl, al),\ al \geq \text{min}(bl, al)$
$x_{u,l} = a_{au,al} \cdot b_{bu,bl}$	$u \leq au + bu, l \geq al + bl$
$x_{u,l} = \text{inv}(a_{au,al})$	$u \leq 1 - al, l \geq 0 - al$

Figure 6.2: Backward propagation constraints

operand can affect less significant bits in the result — when the other operand contains bit-positions less than zero. The presence of fractional bits in one operand causes the bit positions in the other operand to affect bits to the right. For this reason we can only reduce the more significant bits out of a multiplication operand when the least significant bit of the other operand is of a sufficient magnitude.

It is not possible to provide conservative constraints for the inversion operator as not all results of integer inversion can be represented as a finite binary string. As it is necessary for some loss of precision during inversion our choices of constraint are somewhat ad-hoc, and reflect an assumption of the accuracy a programmer will expect from inversion in our problem domain. The problem of modelling inversion more precisely is an interesting open problem for future work.

6.2.1 Analytical Approach

Although the backwards analysis can produce some reduction in the size of variables, it is still a very conservative approximation. There are only certain cases, such as our hypothetical chain of additions, where the rules on necessary information can be applied. The solution that the backwards analysis produces is strictly smaller than the forward analysis alone — but it can only be applied in certain situations and so most computations are still as conservative as the forward analysis alone.

In order to make any serious reduction in the precision required in the program, it is necessary to consider the bias introduced to the computation by the repetition of variables. If any variable is repeated in an expression then the number of possible outcomes in the worst case is reduced. If x and y are of the same magnitude then there are fewer results of $x \cdot x$ than of $x \cdot y$.

Every variable represented at a particular precision can hold a set of bit-strings. This set contains 2^n unique strings for an n-bit value. In our chosen number representation the set of numerical values that the strings map onto is defined by the tuple of the upper and lower bit positions. In Equations 6.1 and 6.2 we define the *floor* and *ceiling* of a variable. These are the minimum and maximum numerical values in the represented set. Equation 6.3 defines the *unit* of the set, or the smallest difference in value between its elements. Finally, Equation 6.4 defines the *value set* of a variable, the represented numerical values in terms of these more basic parts.

$$\begin{align}
\downarrow (X_u, X_l) &= -2^{X_u} \tag{6.1} \\
\uparrow (X_u, X_l) &= 2^{X_u} - \underset{\longleftrightarrow}{X} \tag{6.2} \\
\underset{\longleftrightarrow}{(X_u, X_l)} &= 2^{X_l} \tag{6.3} \\
\overline{(X_u, X_l)} &= \{y | \downarrow X + n \cdot \underset{\longleftrightarrow}{X}, n \in 0..2^{u-l+1} - 1\} \tag{6.4}
\end{align}$$

In the previous section we presented a set of inequalities that are used to analyse the bits that are required in each variable during a computation. A more exact justification for discarding bits within a computation can be seen in the sizes of the value sets in a computation.

The forward analysis propagates the number of produced bits through the computation. In an analysis over a domain of value sets, each arithmetic operation on numbers can be lifted to an operation on value sets. The lifted operator no longer produces solutions to specific equations. Instead finite binary strings are abstracted into sets of possible numerical values. Value sets are constructed directly from the precision specification of a variable; the bit positions within a finite binary string. Equation 6.4 performs such a construction.

Another method of constructing a value set is by applying one of the operators in the numerical domain to a pair of value sets. As the operators are defined over numbers they must be lifted onto the domain of value sets. An example of lifting the numerical operator onto sets of values is shown in Equation 6.5. The syntactic structure is identical for the other operators under consideration, only the specific operation applied to the cartesian product of the two sets varies. The lifting of operators from the domain of finite binary strings to value sets is an abstraction.

$$\overline{X \cdot Y} \;=\; \{x \cdot y | \forall x \in \overline{X}, \forall y \in \overline{Y}\} \tag{6.5}$$

Applying a lifted operator to a pair of value sets is performing a forward propagation on the computation. The set produced is the worst case assumption of all of the possible values that two input sets can produce. When the input sets are exact the resultant value set will be an exact representation of the values an operator can produce. In order to map this value set onto a variable precision it is necessary to find the smallest representation for the elements of the set.

Finding the representation for a value set is the minimisation problem shown in Equation 6.6. Each value set V can be mapped onto many representations, but the desired mapping is onto the representation with the minimal number of bits.

$$\min_{\overline{X} \supseteq V} \quad \{X_u - X_l\} \tag{6.6}$$

A desirable method for analysing precision in a computation is the explicit calculation of the value sets for each variable. Solving the minimisation problem to determine the smallest variable size for each value set would then produce an optimal analysis of the precision. However this approach is not tractable. Applying an operation to a pair of value sets A and B produces a value set with up to $|A| \cdot |B|$ elements. Each of the input value sets has 2^n elements where n is the number of bits in the input. If there are s bits of state in the filter and we assume that the value set for each variable is computed independently, the worst case for a computation of m operations is $2^{(n+s)\cdot m}$ elements in the final value set.

A filter may contains 10's or even 100's of operations, and ideally we would like to analyse input sizes of 8, 16 or even 32-bits, however it is not tractable to compute sets with the order of 2^{3200} elements. This worst case complexity arises from the assumption that each variable (and hence value set) is independent from the others. In reality not every variable will be independent, and it is simple to see that in a system with n-bits of input and s bits of state there cannot be more than 2^{n+s} elements in the final result.

The huge difference between 2^{n+s} and $2^{(n+s)\cdot m}$ arises because in a filter there are many more intermediate values than inputs during each iteration. Each computed value has some contribution from the inputs, when these computed values are combined in further operations there is a bias in the output that results from these identical contributions. As a simple example, consider the operation $X = Y + Z$ where Y and Z are independent. The size of $\overline{X}$ is bounded by $\overline{Y} \cdot \overline{Z}$ where the exact size is dependent on the overlap in the range of Y and Z. In contrast in the operation $X = Y + Y$, the size of $\overline{X}$ is bounded by $\overline{Y}$. The bias that occurs is that each Y in the expression must take the same value. There are only $\overline{Y}$ combinations of two variables Y, while there are $\overline{Y} \cdot \overline{X}$ possible combinations for a pair of independent (or unbiased) variables.

Restricting the complexity of the analysis by incorporating information about the bias between variables requires the final value set to be computed in a single step. Although the analysis results are significantly tighter than the simple propagation rules in the previous section, the complexity is now proportional to the number of values in the computation. Using simple propagation it is proportional to the number of operations, independent of the input size.

Improving the efficiency of the analytical approach is difficult. In order to reduce the number of values stored in each set it is necessary to analyse how the operators distribute over each other, masking out possible values. As the operators under analysis include both addition and multiplication this would entail solving which values can be produced by combinations of multiplication and addition — without enumerating and collecting the results of those multiplications and additions. Such a result is considered to be hard, and would have enormous impact in number theory. Using such an analysis on a single multiplication operation would provide the prime numbers in that range by enumerating which values are missing from the produced set. This problem is too difficult to tackle directly and as such we attempt to solve the precision analysis without addressing it.

Our approach stems from the observation that the technique to find the contents of value sets must try and maintain a formulation of how the previous value sets constrain the contents of the current set. An efficient mechanism to implement this constrained enumeration would attempt to merge these constraints whenever possible, and to delay enumeration of results until all available operations on the constraints have been performed. This approach is very similar to the tabling mechanism used in Constraint Logic Programming (CLP), and there exists a body of well developed CLP solvers which have been extensively improved over many years.

We use a Program Transformation approach to solving the Precision Analysis problem. The precision problem is translated into an equivalent CLP problem, and the program produced is applied to a CLP solver. The solution of this program using the CLP solver contains the necessary precisions for each original program variable. By constructing a CLP formulation of the problem on value sets we are still attempting a search of the 2^n problem space — but using the well tuned heuristics in the solver to prune that space into a tractable search area.

6.3 CLP Transformation

The original form of a program being analysed is a set of independent equations, with type declarations on the input, output and stateful variables. To illustrate the transformation we present the simple example in Figure 6.3. The example is the smallest which contains all of the features of the transformation.

The first stage in the transformation is to order the equations into a valid sequence of execution. Chapter 5 describes these constraints and how they are formed. For the purposes of our analysis, the most important feature of the equations is the partition of the variables into those with a known precision, and those which will be inferred. In our example each of the declared variables, `a`,`b` and `c` has a known precision. The undeclared variables, `t` and `t2` both have an unknown precision, and the implicit variable that holds the intermediate result `a*b` is also of unknown precision.

```
input<3:-4> a;
state<3:-4> b;
output<3:0> c;
b = b - a*b;
t = a+b;
t2 = a*b;
c = t2 - t;
```

Figure 6.3: Example Source Program

```
b = b - a*b;
c = a*b - (a+b);
```

Figure 6.4: Example Expression Trees

For the transformation into a CLP problem these equations need to be merged into expression trees. The expression trees must have the following properties, each variable at:

- the root node must have a known precision
- the leaf nodes must have a known precision
- an internal node must have an unknown precision

These properties control how the equations are merged into expressions. Where an equation contains a variable that is not of known precision, there is always a second equation in which the variable appears on the left hand side. The unknown variable in the first equation is substituted for the right hand side of the second equation. These substitutions are only performed into equations which have a known variable on the left hand side. The substitution process finishes when each equation of a known variable only consists of terms of known variables. The equations of unknown variables are then discarded. Figure 6.4 shows the result of applying this first stage to the example program.

After transforming the source program into individual two-operand statements the program is a set of expression trees. Each expression tree has a known root node, and known leaf nodes. Although the trees can overlap, so that nodes are shared, each tree has the necessary properties that the external (root and leaf) nodes are known, while the internal nodes are unknown. The precision problem is to find the value of each internal node in the tree.

The most straightforward conversion of the precision problem into an instance of a CLP problem would appear to be CLP over the rationals — CLP(Q). In this formulation the program variables become the domain variables that the solver uses. Each of the values that a finite binary string can represent is represented directly as a rational value in a domain variable.

The problem that is immediately encountered is that rational solvers , CLP(R) and CLP(Q), model infinite sets of values in their domain variables. So if a solution is constrained to be between two bounds then there are an infinite number of solutions that can be enumerated. The precision analysis problem attempts to find a minimal representation of the rational values based on the restrictions of which rationals cannot be produced. The restrictions on which rational values cannot be formed are caused by the discretisation of the rational numbers onto a finite domain. Unless the problem model captures this discretisation it will not converge to the desired solution.

Although the problem that we are modelling operates upon rational numbers, the rational representation is not the most accurate way to model the problem. The constraints that can be placed on a variable in CLP(Q) allow the value to be bounded above or below, but not to restrict the granularity between possible values in the set of solutions. There is no way to directly encode the unit size of a variable as a constraint on the problem solutions.

We encode each of the finite binary strings as an integer and model the problem in CLP(FD). In the *finite domain* scheme for CLP each variable holds a finite set of integers. Constraints are posted for the solver as relations between domain variables. In order to encode the constraints between precision levels it is important that we can use $\min, \max$ and modulo constraints. When a constraint is posted between values that are held at different precisions we perform manual scaling of the values to map them into each other's representations. Multiplication and division of values act as scaling operations to transform how each string is encoded as an integer.

To encode a particular finite string it is necessary to scale the represented value to be an integer. For each string this integer value is the most direct encoding; $([x_0, x_1, ...x_{n-1}], u, l)$ is scaled as $\sum_{i=0}^{n-1} x_i \cdot 2^i$. The effect of encoding each string into an integer for the domain variable in this direct manner is that each finite domain variable operates at a different scale. The unit within each domain represents a value of 2^l in the problem domain.

In order to encode constraints between domain variables it is necessary to scale values in order to map them from one scaling domain to another. Given a pair of domain variables X and Y represented as $([X_0, ..., X_{X_u - X_l + 1}], X_u, X_l)$ and $([Y_0, ..., Y_{Y_u - Y_l + 1}], Y_u, Y_l)$ a scaling from $X \mapsto Y$ is constructed as:

$$X \cdot 2^{Y_l - X_l} \mod 2^{Y_u - Y_l + 1} \tag{6.7}$$

The reverse mapping $Y \mapsto X$ is:

$$Y \cdot 2^{X_l - Y_l} \mod 2^{X_u - X_l + 1} \tag{6.8}$$

When mapping between two domain variables it may be the case that one is more precise than the other, for example if $X_l < Y_l$ then X is more precise than Y as the unit of the domain for X is smaller. The scalar that is applied to Y as part of the mapping to X will be fractional as $2^{X_l - Y_l} < 1$. These fractional values cannot be represented as integers in the finite domain and so we use an integer division in the mapping:

$$\begin{aligned} & Y \mapsto X : \\ & X_l < Y_l \quad \rightarrow \quad X = Y \cdot 2^{Y_l - X_l} \mod 2^{X_u - X_l + 1} \\ & X_l \geq Y_l \quad \rightarrow \quad X = \frac{Y}{2^{Y_l - X_l}} \mod 2^{X_u - X_l + 1} \end{aligned} \tag{6.9}$$

The scaling that is applied maps values from one representation, at a specific level of precision, to another representation, with a separate level of precision. The modulo models the truncation that occurs when a variable has too few bits to store a value. This scheme for generating constraints between variables models the fixed point arithmetic. The operations are equivalent to those that would be performed at run-time in a conventional fixed-point implementation. The multiplication and division is usually replaced with shift operations for efficiency. The modulo is implemented as a mixture of logical `and` operations to mask values and the implicit truncation that occurs during storage.

Each of the operations in the expression tree uses three operands; two sources and a target. We generate a constraint for each of these three-operands codes using the scaling scheme. Consider the operation $X = Y + Z$. In the multi-precision arithmetic the result of the addition operation is calculated to an arbitrary precision, and then the result is stored in X. Either X has a known precision, which can cause a truncation in the result, or X is of unknown precision in which case the value is stored without truncation.

The intermediate result is calculated at a precision large enough to avoid truncation. This precision is the conservation approximation that is provided in the forward propagation. For the addition operator this yields an intermediate $([...], \max(Y_u, Z_u) + 1, \min(Y_l, Z_l))$. The constraint generated for the operation scales the two source operands onto this intermediate precision, then scales (and possibly truncates) the intermediate onto the precision of the result variable. Substitution of the intermediate precision into the scaling in Equation 6.9 produces the constraint shown in Equation 6.10;

$$\begin{aligned} Y &\mapsto Y' : \\ \min(Y_l, Z_1) < Y_l \quad &\rightarrow \quad Y' = Y \cdot 2^{Y_l - \min(Y_l, Z_l)} \\ \min(Y_l, Z_l) \geq Y_l \quad &\rightarrow \quad Y' = \frac{Y}{2^{Y_l - \min(Y_l, Z_l)}} \end{aligned} \tag{6.10}$$

Examination of the guards in Equation 6.10 show that the second predicate can never be true. For the division scaling to be used it must be true that $\min(Y_l, Z_l \geq Y_l)$ which never holds as $\forall x, y : \min(x, y) \leq y$. This allows a simplification of the constraint as the division scalings can be ignored between the source operands and the intermediate. Substitution of the scaled variable constraints into the scaling mapping in Equation 6.9 and applying the relevant operation to the variables

```
generateCon( plus, (X,Xu,Xl), (Y,Yu,Yl), (Z,Zu,Zl) ):-
    nonvar(Xu), nonvar(Xl),
    nonvar(Yu), nonvar(Yl),
    nonvar(Zu), nonvar(Zl),
    min(Yl,Zl,L),
    YScale is integer(2**(Yl-L)),
    ZScale is integer(2**(Zl-L)),
    Modulo is integer(2**(Xu-Xl+1)),
    (Xl < L ->
        Scaling is integer(2**(L-Xl)),
        X #= (Y * YScale + Z * ZScale) * Scaling mod Modulo
    ;
        Scaling is integer(2**(Xl-L)),
        X #= (Y * YScale + Z * ZScale) / Scaling mod Modulo
    ).
```

Figure 6.5: Constraint Generator For Addition Operator

produces the constraint in Equation 6.11:

$$\begin{array}{rcl} \text{true} & \rightarrow & Y' = \dfrac{Y}{2^{Y_l - \min(Y_l, Z_l)}} \\ \text{true} & \rightarrow & Z' = \dfrac{Z}{2^{Z_l - \min(Y_l, Z_l)}} \\ X_l < \min(Y_l, Z_l) & \rightarrow & X = \dfrac{Y' + Z'}{2^{\min(Y_l, Z_l) - X_l}} \mod 2^{X_u - X_l + 1} \\ X_l < \min(Y_l, Z_l) & \rightarrow & X = (Y' + Z') \cdot 2^{\min(Y_l, Z_l) - X_l} \mod 2^{X_u - X_l + 1} \end{array} \tag{6.11}$$

As this example shows the constraints placed between variables rapidly become unwieldy because of the disjunctions within each scaling. Despite this complexity the formulation of the constraint generator retains simplicity. The structure of each constraint contains arithmetic operations that are supported in CLP(FD) and a logic structure that partitions each constraint into a set of mutually exclusive choices. The logical structure is expressed directly in Prolog, and the correct arithmetic constraints are posted by evaluation of the SLD tree.

Each condition within the constraint generated for an operation is predicated on the exponents evaluated by the forward propagation. It is unnecessary for these exponents to be an accurate bound on the precision of the variable — but they must be a conservative approximation as they define the encoding into integers of each value set. The conditions can all be checked at run-time, either the exponent estimates in each case are grounded Prolog variables or forward propagation can be applied to grounded values to produce a ground exponent estimate. None of the domain variables in the constraints require evaluation, each usage is a definition of the relationship between domain variables.

Each of the partial constraints for a particular variable in Equation 6.11 is guarded by an exclusive condition. There are no cases where multiple definitions of the same variable can occur. This allows a straightforward formulation of the system in Prolog so that no backtracking is required to post the constraints for the solver. Although the solver will use backtracking internally to solve the problem it requires a single complete state to be posted before execution.

The implementation of the generator for the addition operation is shown in Figure 6.5. The guards at the beginning of the predicate ensure that this body is only called when the exponent estimates are known. The only disjunction left in the generator after simplification of the constraint cases is how to apply the final scaling; either through multiplication or division. Each program variable is represented in the implementation as a tuple of three Prolog variables. The first variable in each tuple is the CLP domain variable. These variables are not directly manipulated by the Prolog program but act as references to domain variables in the CLP solver. They can still be compounded into, and pattern matched out of, Prolog terms for storage during execution. Any attempt to unify these references will destroy the connection to the CLP solver's internal state.

The other clauses of generateCon are structured in the same way. The only difference in each case is the operation applied to the scaled source operands. The analysis of a domain program proceeds as follows:

1. Convert program into three-operand form.
2. Perform forward propagation to determine initial exponent estimates.

3. Define a CLP domain variable associated with each program variable.

4. Call generateCon once for each tuple of domain variable and estimates.

5. Call the solver.

The interface to the CLP(FD) solver allows bounds to be extracted for each domain variable. The minimisation problem in Equation 6.6 is solved for the upper bound of the domain variable. The solution to this problem is the number of bits required to encode the set of integers that the solver has computed. In combination with the estimated lower bit position this result produces the required upper bit position for the variable. There are two methods to extract the correct lower bit position. The least accurate, but most efficient method is to simply use the lower bit position estimate. As the estimate was conservative this value will be a correct bit position for the variable, although it may use more bits than are necessary.

The less efficient, but more accurate method of computing the lower bit position is to partition the solution set into subsets with common values for the lower bits. If the lowest bit in the variable can be removed without affecting the represented value then all of the integers in the solution set must be even. Even integers have a zero as their lowest bit, which matches the definition of bits not explicitly defined in the finite binary string. Furthermore, if all of the integers in the solution set are 0 mod 4 then their lowest two bits can be removed. It would be desirable to partition the solution set according to the solution modulo some constant, and then determine which bits could be reduced by which partitions contain values. This is not currently possible with a CLP solver as the following session shows:

```
?-  domain([A],0,7), B#=A+A, C#=B mod 2.
A in 0..7,
B in 0..14,
C in 0..1 ?
```

The bounds on the resultant variable `D` are not tight enough to determine that the partition for 1 mod 2 is empty. It is not clear that future CLP solvers could tackle this problem as finding the solutions of arbitrary equations congruent to a constant is a known hard problem in cryptography. Instead we must explicitly enumerate and test the integers in the solution set. The complexity of this procedure is linear in the size of the set or $O(2^n)$ where n is the number of bits in the result. This makes it feasible only for small data-sets, otherwise we must fall back to the less precise method above.

Experiments using the CLP(FD) solver in Sicstus showed that the analysis is practical for programs in the DSL we have presented. As the bit-size of the variables increases beyond 8-bits the runtime of the solver increases rapidly, becoming intractable beyond a 16-bit system.

6.4 Conclusions

In this chapter we have investigated the static analysis necessary to define an efficient executable version of a program written in a DSL. The first technique used a system of constraints that model the propagation of precision between variables in a program DSL. The system can be implemented as a data-flow analysis, propagating information forwards, and backwards, through the program until a stable fixpoint is reached. This method uses the same set of constraints applied in previous work in the literature as the forward propagation pass. The addition of the backward propagation pass can produce results that are strictly more precise than previous work — however the application of the backward pass is limited in practice and only delivers large improvements in the toy examples given. The difficulty here is defining a correct set of constraints on which bits are necessary to correctly define an output result.

The second technique investigated in this chapter uses meta-programming to produce a CLP representation of the program. The CLP implementation can then be executed by a CLP(FD) solver. This approach builds on decades of research into CLP, by applying the body of CLP(FD) techniques to the precision analysis problem. The novelty in this area of the work is transforming the problem into a form that a CLP(FD) solver can operate on. Similar work exists in applying CLP to symbolic analysis of programs [66] but not to the precision analysis problem.

Our work demonstrates that constraints can be mapped onto a logical representation of the program, and that such a representation can be executed to find relevant precision. The technique is at least as precise as the data-flow technique, and can be much more precise in some situations. However the approach is not tractable for analysing systems with a large input size, or a large state. It is unclear that the technique could be made practical as the runtime is an exponential function of the size of the input / state. It can be considered unlikely that advances on the problem will be incorporated into CLP solvers, and the problems can be mapped onto known hard problems in number theory.

Despite the poor complexity of the CLP approach, the extra precision that it affords over the data-flow approach makes it attractive for small problem instances, such as the micro-controller that we have studied.

In the next Chapter we shall examine the transformation of programs from the DSL into an executable form. The transformation is independent of the analysis used; it is simply assumed that the precision of the source program is provided. For this purpose either of the two techniques presented could be used, or an existing technique from the literature.

Chapter 7

Multi-precision Code Generation

Chapter 5 presented a domain-specific-language (DSL) for writing filter programs. The programs operate on values that have the properties of numbers. The semantics of the language allow the programmer to write programs that directly manipulate numbers without having to worry about how precise the representations of those numbers are. The language semantics guarantee that the compiled program will be numerically correct. None of the inferred variables will introduce roundoff errors into the program results.

The numerical analysis in Chapter 6 produces a precision annotation for each variable in the program. The resultant program specifies multi-precision arithmetic that is parameterised to a specific precision. In this chapter we present a method of compiling this parameterised multi-precision arithmetic into an efficient program for the target device.

Our method is a program transformation approach. An interpreter for the DSL is written. It is our claim that the structure of this interpreter is a clear specification of how to implement the language — more so than a conventional compiler. A compiler is then generated automatically using a specialiser. The specialiser transforms a program to make it specific to a particular input. The novelty in our method is the language that each program is implemented in, and operates upon. The specialiser is a generic tool that operates upon a meta-language. It has no knowledge of the language being compiled, or of the target language it is compiling into. Our method transforms programs from the source language into the target without an explicit definition of this mapping.

7.1 Overview of Specialisation Process

The target domain that we have used within this thesis has tight constraints on energy usage. To fit within these constraints micro-controllers have been used; each of these processors has a limited instruction set, little data memory and a small program store. The low-level of this target domain means the only practical programming method is code written directly in assembly language.

From the designer's viewpoint, the filter is a set of equations defining the relations between input and output. The equations are composed of simple arithmetic operations but their semantics are defined over arbitrary precision bit-strings. This abstract viewpoint is a long way from the actual implementation as a series of micro-controller instructions.

The problem is compilation of a DSL program into this low-level representation. The program will have been analysed, using the methods in Chapter 6, and each variable annotated with a static precision. This precision specifies exactly which set of rationals can be stored in the variable, and which bits they are encoded into. The sequence of instructions that implements each arithmetic operation in the program is dependent upon these annotations. Each operation is sensitive to the context in which it is used. The annotations of the operands that a pair of identical operations apply to may result in two different instruction sequences.

The goal of the research is a method of performing this context sensitive transformation without covering each of these possible outputs manually — both an exhaustive and exhausting approach. Writing a conventional code-generator would require each combination to be broken down, and code written that outputs the correct sequence of operations. Our approach is to write a single interpreter for the DSL, a program that implements operations in the chosen numerical domain. It is still necessary to check the context that each operation occurs within — but now this becomes a problem of performing different instructions in different states.

The difference between the compiler construction and the interpreter construction is an extra interpretive layer in the former case. While a compiled program is more efficient than an interpreted one, this is not true of the program transformers. A layer of interpretation is removed by compiling a program; the residual from the interpreter executing the source program removes some of the interpretation. But an extra layer of interpretation exists in a compiler in comparison to an interpreter. The compiler cannot just interpret the steps in the source language, it must encapsulate these steps and produce a record of them as an executable program.

To solve our problem we demonstrate a method of writing an interpreter for our high-level DSL and using it to automatically generate a compiler. Automatic generation of efficient compilers from language interpreters is a long-standing goal of the partial evaluation community. Our small contribution towards that goal is to show a mapping between two disparate languages using a meta-language. The meta-language acts as a glue between the two domains; the target domain is too restrictive to implement an interpreter of the source domain. Another benefit is that we only use program transformation tools on the meta-programming language — there is no need to use a specialiser tailored to the DSL.

The transformation of a program from the source language to the target language is an instance of the first Futamura Projection [21], shown in Equation 7.1. This equation does not specify the implementation language of each program. The assumptions are that the program being compiled P is written in a language that the interpreter int can execute, and furthermore that the specialiser $spec$ operates on the implementation language of int.

$$\forall i : [\![[\![\, spec]\!](int, P) \,]\!](i) = [\![P]\!](i) \tag{7.1}$$

These assumptions on the implementation language cannot be made for our problem. We are attempting to construct an interpreter of a domain language (D). The interpreter can then be partially evaluated with a program in D as static input. The evaluation compiles the program into the language that the interpreter is written in. Ideally this interpreter would be implemented in the target language (T), and the compiler would be complete as shown in Equation 7.2, a parameterisation of the first Futamura projection.

$$\forall i : [\![\ [\![spec_T]\!](int_T^D, P_D)\ ]\!](i) = [\![P_T]\!](i) \tag{7.2}$$

This construction would be sufficient if D and T are suitably related. However, we have deliberately chosen T to be a very low-level language — the machine code of a micro-controller. Furthermore, we have chosen D to be as abstract as possible in order to automate as much of the compilation as possible. Because the definition of T is bound tightly to the implementing architecture, there is a problem in attempting the straightforward translation. T is not rich enough to support such an interpreter of D. It is not possible to construct int_T^D.

Our solution to this problem is to use a language that is syntactically isomorphic to the target language. Programs in this language can be converted to the target language by a simple syntactic substitution. In order to explain this language and isomorphism, we assume that int_T^D exists, and that we have a rich meta-language (M) that can be used as an intermediate.

A second interpreter can be written, int_M^T, that implements the semantics of the target instruction set in the meta-language. The second interpreter can be thought of as an emulator of the target architecture. The first interpreter is implemented in the language that the second interprets. Thus, the two can be composed together so int_M^T executes the instructions of int_T^D which executes the instructions of the target program.

Partial evaluation of the first interpreter with respect to the second will produce an interpreter of D that is implemented in M. The structure of this generated interpreter will retain the semantics of the target language (T). The semantics of the T instructions to interpret each step in D will be left in the residual code, while the interpretive overhead of executing D will be reduced from the residual interpreter. Syntactically, the constructs that execute the instructions of T will be left in-place within the residual interpreter.

The composition of the interpreters is shown in Equation 7.3 where we term the language of the final interpreter M'. This language is a restriction of the meta-language M' such that operations in M' are encoded in pieces of int^D_T that map directly onto the target instruction set. Programs in M' have all of the expressive power of M, but the parts of them using the predicates that model T are syntactically isomorphic to programs in T.

$$[\![spec_M]\!](int^T_M, int^D_T) = int^D_{M'} \tag{7.3}$$

We may now discharge our assumption; although we cannot implement int^D_T directly we have manually created both int^T_M, and $int^D_{M'}$. These are the target emulator and the DSL interpreter respectively. The handwritten DSL interpreter calls the instruction implementations in int^D_M directly. If the interpreter is written with some care, then when specialised against a program in D all of the operations which are not syntactically isomorphic with T are correctly analysed as static and removed from the residual. Ensuring that all calls to isomorphic predicates remain within the residual is easier, as we have a single dispatch point within the target interpreter that we can explicitly annotate as $rescall$ (a residual call) in the BTA.

One advantage of this manual construction is illustrated by our compilation process in Equation 7.4. The only external tool that we require is a specialiser for the meta-language, there is no need to write any analysis or transformation tools that natively understand either the source language (D) or the target language (T).

$$\forall i : [\![\ [\![spec_M]\!](int^D_{M'}, prog_D)\ ]\!](i) \cong [\![prog_T]\!](i) \tag{7.4}$$

A predicate written in M' is *syntactically isomorphic* to a program in T, denoted $\cong$, iff it contains a conjunction of terms that are syntactically isomorphic to T. A term is syntactically isomorphic to a program in T iff it is a call to a predicate in the target interpreter or a call to a predicate that is itself syntactically isomorphic to T. More informally, when a program in the meta-language reduces to one that consists only of calls to predicates that model the instruction set of the target, then the residual program is syntactically isomorphic with a target program as we can perform a simple syntactic substitution to rewrite it as PIC assembly code.

Both the DSL (D), and the target (T) are described in Chapter 5. In the remainder of this chapter, we describe the features of the languages relevant to compilation, and the interpreters required to implement this approach. Section 7.2

$$\text{bit}(n, \text{var}(u, l, bs)) = \begin{cases} s & n \geq u \\ bs[n-l] & l \leq n \leq u \\ 0 & n < l \end{cases}$$

$$\text{where} \quad \begin{aligned} s &= bs[u-l] \\ (x_0, x_1, ...)[n] &= x_n \end{aligned}$$

Figure 7.1: Definition of bit-positions within a variable

describes the DSL implementation. The relevant features of the target (T) are described when it is embedded into M' in Section 7.3.

7.2 Domain Specifi c Language — D

The DSL and variable format is described in Section 5.1. After the analysis described in Chapter 6 is applied to a program in the DSL every variable is represented as shown in Figure 7.1. The value of a bit n in the variable var(u, l, bs) is described by the function bit. As explained previously with slightly different notation, u and l are the most, and least, significant bit positions with bs as a bit-string holding the values of the bits between positions u and l. The DSL interpreter operates directly on this representation of a variable at a known precision.

The most basic function that we define on these values is transformation; mapping from a value represented at one precision, to the closest representation of the value at another precision. Under the chosen rounding mode the closest representation to the value is the representation less than, or equal to the value with the smallest difference. This function is not invertible, as information is lost when mapping from a higher precision to a lower precision. This lost information means the function is not injective, and will map several values onto the same representation in the lower precision.

The transformation function is shown in Figure 7.2. The simplicity of the function definition is a result of the bit being well defined in all positions in the original precision. This results in a lack of corner cases and thus a uniform defi-

$$\begin{array}{rcl} \text{trans}(\text{var}(u,l,bs),ou,ol) & = & \text{var}(ou,ol,obs) \\ \text{where} \quad \forall n : ol \leq n \leq ou & & \\ obs[n-ol] & = & \text{bit}(n,\text{var}(u,l,bs)) \end{array}$$

Figure 7.2: Transformation function

nition.

Using a transformation function makes the definition of individual arithmetic operations simpler. It is no longer necessary to consider the various cases of which bits are contained in a value representation, and which are not. The set of operations that are supported in the language is addition, subtraction, multiplication and inversion.

The implementation of each operation in the interpreter has a common structure:

1. Transform both operands to an intermediate precision that preserves bit-correctness in the output.

2. Perform the operation between the two representations at the intermediate precision.

3. Transform the resultant value to the output precision.

The intermediate precision may be larger than the output precision, for example in the case of addition where carry chains must be computed to determine the correct result. For each operation the intermediate precision with respect to a given pair of input precisions is shown in Figure 7.3.

The table may be compared to Figure 6.1 in Chapter 6. The intermediate precision used for each operation is the worst-case constraint used in the analysis. The analysis removes excess bits from intermediate variables in the computation. Using the worst-case assumption for the intermediate storage within an operation does not cause the size of program variables to saturate as they have been clamped by the analysis stage. The constraints in the analysis are applied between

Operation	Intermediate Precision
$(au, al) + (bu, bl)$	$(\ \max(au, bu) + 1, \min(al, bl)\ )$
$(au, al) - (bu, bl)$	$(\ \max(au, bu) + 1, \min(al, bl)\ )$
$(au, al)\ \cdot\ (bu, bl)$	$(\ au + bu, al + bl\ )$
$1\ /\ (au, al)$	$\begin{cases} al < 0 & (\ 1 - al, \perp\ ) \\ al \geq 1 & (\ 0 - al, \perp\ ) \end{cases}$

Figure 7.3: Intermediate and output precisions

variables to reduce the increase in precision for the variables in a chain of operations. The worst-case assumption here only applies within a single operation and so does not cause this increase in precision through the program. The final result is transformed onto the precision of the result variable.

For addition, subtraction and multiplication it is possible to determine this required precision statically as it specified by the precision of the operands alone. For inversion the required precision is dynamic, depending on the actual value of the operands. There is no safe approximation as the reciprocal of any value that is not a power of 2 will produce an infinite repeating binary string as a result. If we assume that there is no division by zero then the expressions in Figure 7.3 are a safe approximation for the upper bit position. The lower bit position of the result variable is used as the lower position for the intermediate value. There must be some loss of precision when performing division on a machine with finite resources, and the bit position provided by the analysis should be sufficient to provide bit-correctness in the final result.

The pseudo code in Figure 7.4 shows the process for performing each operation in the language. This construction is uniform across both the operators in the DSL and the different precisions at which these operations can be computed. The definition of the finite binary strings has been used to decompose the operation into more basic parts. The transformation function works because the binary string defines the value of each possible bit. This allows the numerical operations to be performed in a uniform way across all the combinations of operand sizes. The problem of generating unique code for each precision context is transformed

```
perform(op,var(au,al,av),var(bu,bl,bv),ru,rl) =
    (iu,il) = Select from Fig.4 based on op
    ai = trans(var(au,al,av),iu,il)
    bi = trans(var(bu,bl,bv),iu,il)
    rs = perform op on ai and bi
    defined(ru,rl) ->
        return var(ru,rl,trans(var(iu,il,rs),ru,rl))
    undefined(ru,rl) ->
        ru = iu
        rl = il
        return var(ru,rl,rs)
```

Figure 7.4: Pseudo code for each operation

into a uniform method of interpreting the program.

The interpreter that we have described could be implemented directly in the meta-language without regard to the isomorphism with the target language. This implementation would be executable and could be used by the designer to develop programs and test their correctness. An implementation of the interpreter as illustrated in Figure 7.4 would implement arbitrary precision arithmetic. There are cases to handle an undefined resultant precision in the pseudo-code that propagate the precision dynamically.

Although such an implementation would be useful to the designer, it is not feasible for executing programs on the target device. When constructing the interpreter $int^D_{M'}$ we can rely on ru, rl being statically known to simplify the interpreter to the common code and defined case in Figure 7.4. When these values are dynamic it is not possible to partially evaluate the decision in each operation. Using static ru, rl values allows the choice to be reduced out and all of the applications of the `trans` function to be specialised to the precision parameters.

The partial evaluation of the operation itself, and the surrounding transformations will produce a specific sequence of target instructions. The mappings between source code / annotations and this target code could have been explicitly written as a compiler. However this executable formulation of the problem is easier to write than something that generates an executable solution to the problem. The implementation of each operation in this manner must produce a residual that is isomorphic to the target language. This process of embedding the implementation in M' is the subject of Section 7.3.

7.3 Embedding The Target Language

The meta-language (M) is Ciao [31] (a dialect of Prolog). Prolog is used because of the maturity of analysis and specialisation tools for the language and their state of integration into Ciao — in the form of Ciaopp [**?**]. One of the properties of our compilation technique is that the chosen specialiser will not explicitly manipulate programs in D or T. This property ensures that we can select a specialiser according to its maturity rather than restrict ourselves to the specialisers that have been implemented for the languages we are compiling between.

The choice of language for M must be well-suited for the construction of the target interpreter. Most of the operations within the interpreter are mappings from one domain to another. These maps have a natural form as logical predicates. Prolog is both typeless and symbolic with a simple encoding of anything as data (even predicates) that allows a rapid cycle of formulating ideas and testing them as code. An important feature of Prolog is that programs are typeless. When writing code at several different language levels it is possible to test the parts in isolation without constructing a correctly typed framework for them.

Each instruction in the target language maps to a predicate. M' is the combination of Prolog with calls to these predicates. The implementation of these predicates is the target interpreter int^T_M. The interpreter can be thought of as a library which provides the semantics of the target instructions to Prolog programs. Each predicate transforms an initial state to an output state as directed by its parameters. An example is given in Figure 7.5. The *addwf* instruction retrieves two values from the initial memory state (one from the working register 0, and one specified in the first parameter), adds the two values, and then depending on the

```
addwf(S,F,D,S3) :-
    lookup(0,S,W),
    lookup(F,S,X),
    Ans is W+X,
    AnsM is Ans mod 256,
    (D=:=1 -> store(S,F,AnsM,S2)
            ; store(S,0,AnsM,S2)),
    statusBits(Ans,S2,S3).
```

Figure 7.5: Sample implementation of T in M

```
mTransAd( (L,V1), L, (V1->V2), (L,V2) ).
mTransAd( (L,V), L2, (_->_), (L,V) ) :- L \== L2.
mTrans( S, L, X, S2) :- map(S,mTransAd(L,X),S2).
lookup(Ad,(_,S),V) :- mTrans(S,Ad,(V->_),_).
store((D,S),Ad,V,(D,S2)) :- mTrans(S,Ad,(_->V),S2).
```

Figure 7.6: State transition predicates

second parameter, either stores the result in the working register, or in the specified register. The carry and zero flags are set according to the result.

The memory state is represented by a simple list of pairs of integers, one with the location label and one with the current value bound to that label. The lookup and store operations retrieve the value from a label, and produce a new state with the label set to the given value. These operations are expressed clearly in a declarative language as shown in Figure 7.6.

The predicates in Figure 7.6 implement imperative state transitions directly in a logical language. These predicates allow the target instruction set to be expressed clearly in Prolog. In the PIC micro-controller each instruction is located at an address in memory and the semantics of the instruction-set specify how the state transitions affect the PC, indirectly controlling which instruction is executed next. In the target language T a program is comprised of a list of instructions. The control-flow is contained implicitly in the sequencing of the list. When T is embedded within M the control-flow through the instructions is the Prolog control-flow around the calls to the target predicates. In general this can be more complex than a simple sequence, although programs in M' that are syntactically isomorphic to T are a simple sequence of conjunctions by definition.

The interpreter of T implements the semantics of the individual instructions, rather than emulating the machine that executes them. This subtle difference allows Prolog code to be freely interleaved with calls to the target interpreter. The resulting code can be executed natively in M as the explicit threading of the state gives a well defined meaning to the program. If a partial evaluation removes any surrounding Prolog control flow decisions (reducing them to conjunctions) then the resultant code has a syntactic one-to-one mapping with a PIC assembly language program. The syntactic mapping has to relocate the program to an absolute address in the program memory, but as all control-flow in T is relative this is a trivial mapping.

One difficulty this abstract control-flow poses is how to interpret the *skip* instructions that conditionally skip or execute the next instruction. In the PIC with each instruction bound to a location the PC can be incremented to implement skipping. When a program in T is embedded in M there is no direct relationship between one instruction and the next in the control flow. This relation now depends on the Prolog code which calls the two instruction predicates. This difficulty is overcome using the higher-order features of the symbolic meta-language. The entire state is encapsulated within a functor (i.e. $S \mapsto skip(S)$) when the skip condition holds. Then each PIC call is wrapped in a dispatcher that checks whether to execute or skip the predicate that is passed in. There is minimal change to programs written in M', and the interpreter merely requires the addition of a dispatcher as shown in Figure 7.7.

Implementing the target interpreter in M creates a language M' that allows a clear construction of the DSL interpreter. The clarity is achieved through interleaving declarative logic (symbol manipulation, backtracking and query solving) with calls that implement the semantics of instructions in T. The predicates implementing T must be passed ground values for instruction parameters, and must

```
pic(Inst) :- Inst =.. [P,skipping(S),S].
pic(Inst) :- Inst =.. [P,skipping(S),_,S].
pic(Inst) :- Inst =.. [P,skipping(S),_,_,S].
pic(Inst) :- Inst =.. [_,(_,_)|_],
             call(Inst).
```

Figure 7.7: State dispatcher

```
extendSign(S,(_,_,8),S).
extendSign(S,(Ad,Bit,Fill),Sout) :-
    Fill<8,
    OrMask  is 255 - (1<<Fill)+1,
    AndMask is (1<<Fill)-1,
    pic(btfss(S,Ad,Bit,S2)),
    pic(andlw(S2,AndMask,S3)),
    pic(btfsc(S3,Ad,Bit,S4)),
    pic(iorlw(S4,OrMask,Sout)).
alignByteInW((D,M),V,B,Sout) :-
    member(var(V,_,Vu,_,_,_,_),D),
    B>Vu,
    bitAddr((D,M),V,Vu,SignByte),
    aligned((D,M),V,Vu,SignBit),
    extendSign((D,M),(SignByte,SignBit,0),Sout).
```

Figure 7.8: Example DSL interpreter operation

be fully deterministic. When backtracking is used to enumerate possible code generations in the DSL interpreter each execution of an instruction must generate a single transition in the state. Multiple transitions for a single instruction with ground parameters will generate spurious call traces and erroneous partial evaluations.

7.4 Interpreter of D in M'

Using the combination of the target interpreter and the meta-language that form M' it is now possible to write an interpreter for D. We have argued previously that it is not possible to write such an interpreter directly in T. The overhead of interpreting instructions, and maintaining a state for D and its mapping onto the device would consume all available resources.

Writing the interpreter in M' solves these problems. The state of the interpreter is composed of a static precision (upper and lower bit-position) for each variable, and a dynamic value (bit-string). The bit-string requires a mapping between partitions of the string and memory locations on the target device. The mapping can be defined declaratively in M' and passed to the interpreter as a static value. The only remaining dynamic values are the values of the bit-string — these must fit within the program memory for the filter to be executable.

We consider two representations of the bit-string. In a packed representation the lowest eight bits of the string occupy one location, each partition of eight bits occupies a sequential location in memory. In a modulo representation each bit n of a location maps to a position that is $n \ mod \ 8$. These two representations have a trade-off; packed variables take less memory but mod variables require less instructions to operate upon.

In M' we can declare disjoint clause bodies for each representation that map to different calls to target instructions. This method requires no run-time overhead and no extra code in the interpreter as backtracking at specialisation time will select the appropriate clause in each context.

Clarity is the result of operating directly on the same symbolic values in both interpreters. Performing an operation on bit-strings has a simple declaration as a uniform loop over a window of bits. Writing the same operation on locations which hold partitions of the bit-string is more complex as there are more cases to consider. All loops and queries within the interpreter that are dependent on the positions and sizes of bit-strings will be statically unrolled into target call sequences. Depending on the relative alignment of bits between a pair of registers, we must perform a different series of target instructions to implement the DSL

$$\begin{aligned} o = a \cdot b \iff \quad & \forall p \in [o_l..o_u] \; : \; p - o_l \equiv 0 \pmod 8 \\ & carry_{[p+n,p+8]}, o_{[p+7,p]} = a_{[p+7,p]} \cdot b_{[p+7,p]} \cdot carry_{[p+7,p]} \end{aligned}$$

Figure 7.9: Mapping Of Intermediate Binary Operation

operation.

In order to encode each DSL operation as a uniform loop over bits, we require an intermediate operation in the interpreter; alignment. This operation is composed of target calls and declarative logic. We show the simplest case of alignment in Figure 7.8. In this case we are attempting to extract a range of bit positions that are above those stored in the state, we extend the sign of the top bit in the represented value.

The alignment operation uses the variable declarations in the DSL state to find the bit-positions that are stored. It can then map these bits onto the representation in memory. There are several possible cases to be considered, each of which forms a predicate body. The requested bits may be above, below or within the stored bit-string. The representation of the value affects whether the bits are within a single location or span several locations.

Given the alignment operation, each DSL operation is constructed as a uniform loop over bit-positions. The iterations of the loop that access bits outside the string will be specialised away. The structure of a pseudo DSL operation is given in Figure 7.9, showing the uniform operation on 8-bit partitions of the value, regardless of the precision and representation in memory. The $x_{[u,l]}$ notation indicates the bits l to u in the variable x, which are instances of the alignment operation.

7.5 Application

We have presented a formulation of the DSL interpreter that operates on a program, a declaration of the variable precisions and a declaration of variable representations. When this interpreter is specialised with respect to these parameters it produces a residual program in M' that we can convert to PIC assembly language.

The trade-off in choosing how to represent each variable is complex and hard to express. The trade-off is a constraint problem where the set of constraints to solve is specific to each program in the DSL. This constraint problem could be specified as a transformation of the program into a constraint language, but the formulation would be difficult.

As M' is a logic language we can use the declaration of representation to generate possible variable representations for the program. Each of these generated values can be supplied to the interpreter to automatically generate code using the representation. Each of these generated codes can be compared to find a target program that matches a given constraint (eg code size or memory usage). This example shows how the search space of the DSL interpreter can be explored to find solutions (target programs) that match criteria that are difficult to express directly as search problems.

The low code density of the PIC limits the complexity of an operation that we can demonstrate code generation upon. We will use the alignment case shown in Figure 7.8 and an entry point that includes an interpreter state with a single variable. The variable uses a packed representation within the PIC memory, its lowest bit is aligned with the lowest bit in a register location, and its higher bits are spanned onto the next location. This call to the alignment operation is shown in Figure 7.10.

The specialiser removes the static arguments to the call, which include the variable being accessed. The resultant code has eliminated the declarations that access the interpreter state and produced constant values as the parameters to the PIC calls. The variables in the resultant code are renamed in order to preserve sharing, which leaves the state threading intact in the produced code. The output is a straight-line code sequence composed entirely of calls to the PIC predicates, and can be syntactically transformed into the PIC program shown.

The interpreter state and all interpretive overhead has been removed from this operation, as the code does not modify the layout of variables in memory. Supplying different memory layouts, and variable representations produces the appropriate access code. In larger interpreter fragments (such as multi-precision addition

```
:- entry alignByteInW(( [var(x,packed,6,-4,10,2,_)],S ), x, 7,
                      ( [var(x,packed,6,-4,10,2,_)],S2)).

alignByteInW(A,E) :-
    pic(btfss(A,11,1,B)),
    pic(andlw(B,0,C)),
    pic(btfsc(C,11,1,D),
    pic(iorlw(D,255,E)).

PIC program:
    BTFSS   11,1
    ANDLW   0
    BTFSC   11,1
    IORLW   255
```

Figure 7.10: Specialisation of getSign predicate

and multiplication) there is a more complex trade-off between the choice of representation, and the size of the residual code.

An alternative approach to implementing the DSL on the target device would be to write a library performing multi-precision operations. The contrast with our approach is dramatic, both in terms of complexity and the efficiency of the final code. The declaration of the alignment operation defines a series of condition checks and an appropriate body to execute in each specific context. This overhead is removed entirely by the specialiser.

Writing each DSL operation as a library function would require a set of cases that contain all of the used calls (eg 8-bit / 16-bit). Each case would need to be written separately, increasing the complexity of the implementation. Any unused bits in the computation, such as using an 8-bit multiplier on a pair of 6-bit values would introduce unnecessary overhead and the performance of the program would suffer. Our approach avoids this problem by using the specialiser to automatically determine which expansion is necessary in each context. A library approach would also require the precision for each variable to be passed to the routine. In contrast our approach reduces these values to constants, and they are implicitly defined when needed so they require no storage. The disadvantage is that unrolling the code in this way increases the program size.

7.6 Related Work

The approach of code generation by abstract operations is detailed in *delayed code generation* as proposed by Piumarta [52]. His method of producing object code for a smalltalk compiler uses intermediate operations between the source and target languages that contain abstract values. These values can be operated on through *deferred* operations that can eliminate intermediate steps in the object code. This approach differs from our method in that these operations are performed upon syntax trees within the compiler. We execute our operations directly through a partial evaluation and reduce expressions based on the static analysis performed within the specialiser.

The uses of partial evaluation have been studied extensively [27, 34], and in particular the use within meta-programming to compile domain specific languages [33]. The novelty of our approach in comparison to these techniques is the construction of the domain interpreter within an extension of the target language. This extension allows us to use the expressive power of the declarative meta-language to write our interpreter clearly and simply. The elements of the target language are residualised during the specialisation, whereas the operations in the meta-language are entirely static. This produces a syntactic isomorphism between the residual program and the target language.

The connection between macro languages and multi-stage computations has been investigated in a functional setting by Ganz et al [24]. They formalise their macro language (MacroML) as a MetaML interpreter and use the features in MetaML to show that their language is type-safe (in that macros cannot create type errors in the final program) and stage-safe (in that macro expansion does not rely on run-time evaluations).

The work on MacroML differs from our own in that we are using an untyped language as the meta-language, and rather than explicit staging constructs we are using online partial evaluation to separate the stages and perform the macro computation. Our work focuses on how to use an implicitly staged computation to perform macro operations; these are not limited to the operations in a conventional macro language as we can freely mix data values between the stages, allowing some runtime values to effect compile-time computations. Of course these runtime values are restricted by the static analysis used within the specialiser.

The approach in Sh [45] uses static meta-programming to embed a shader language in C++ templates. The shader programs are constructed at run-time by recording the operations executed in the meta-language. This reuses the bulk of the C++ compiler for parsing, grammar checking and code generation. This approach is similar to our embedding within a host compiler, although the recording of operations is performed at partial-evaluation time by the specialiser leaving no runtime overhead.

Previous work in compiling embedded languages [18, 63] has looked at embedded typed languages within an existing typed functional language. This allows the meta-language to act as a host compiler and produce code for the DSL. This differs from our approach as we are embedding a lower level untyped language that contains the semantics of the executable target domain. In particular, the construction of our interpreter is similar to the handwritten cogen of [18], in that we have deliberately written the output of a theoretical specialisation. We cover this aspect of the work in Section 7.1.

Herrman and Langhammer show a similar approach to our work in [10]. A DSL for image processing operations is constructed for a similar class of filter programs. Efficiency is also a concern in that domain and their system generates code from an interpreter to remove the interpretative overhead. However, their interpreter uses explicit staging constructs rather than a specialisation approach.

7.7 Conclusions

In this chapter we have shown a method for compiling a high level symbolic DSL into an efficient executable form for an extremely constrained target device.

We have shown that embedding the target language in our choice of meta-language allows the construction of an interpreter that would not be feasible otherwise. This interpreter is both simple, and clear, yet when specialised it produces a residual program that is syntactically isomorphic with with the target language. This achieves compilation from the domain language into the target language by a specialiser that only operates in, and on the meta-language.

Chapter 8

Conclusions and Future Work

During the history of computer science there has been considerable progress in the development of computer programs. Much of this development has been a result of raising the level of abstraction that is used in writing software. The ability to construct large scale software projects is dependent on the tools and methods to manipulate these layers of abstraction in order to produce the low level code that is executed by a computer.

While the benefit of more abstract languages is an increase in the productivity of programmers, the cost of such abstraction is a decrease in the efficiency of the final code produced. In most domains that decrease in efficiency has been offset by an exponential increase in the performance of computer hardware. The same trends that have caused this increase in performance have also made it possible to construct smaller, cheaper, and more power efficient hardware.

Embedded computing, like general computing, is a field in which each generation of processors is more sophisticated than the last. However, unlike general computing, the performance of the state of the art processor remains largely constant. This is because each advance in process technology can be used to make the same level of performance available in a smaller package, at a lower cost, and with a lower energy requirement.

The applications that drive embedded computing seek a ubiquitous usage of computing. This demand constrains the increasing performance that is available.

For this reason the methods and tools for software creation in embedded computing have not progressed at the same rate as in other fields. Methods of software development in embedded systems have depended on a style of low-level development that has been long forgotten in other fields. In the extreme, people are still required to program directly in assembly languages for processors that are not powerful enough to execute code from a higher-level language.

In this thesis we have attempted to increase the ease and efficiency of software development in embedded computing. The results have been promising as they have shown that it is possible to aid the development process at such a low level of abstraction. But a pattern of diminishing returns is visible in the results. Each result has increased the precision at which a particular side-effect of executable code can be analysed. However the complexity of each result is much larger than previous results in the literature. This suggests that further increases in precision will be costly in terms of the complexity of the analysis required.

8.1 Timing Effects

The first side-effect that we have investigated is the timing behaviour of programs. The resultant Timing Analysis is more precise than previous work in several ways. The structural analysis of programs performs a more precise analysis of the loops within programs, allowing the analysis of a larger class of control-flow graphs than previous work. The cost for this increased precision is a loop detection algorithm that is more complex than previous work.

The abstract interpretation that we have shown produces a fine-grained set of timings between program locations. Rather than compute an upper bound on this set, as in WCET, we can produce a conservative approximation of the structure of the set itself. This increase in precision has several applications which would make for interesting future work.

Security One method of attacking the security of embedded devices such as smart cards is a timing attack. Using the information that different code paths in the smart card software take different lengths of time to execute it is possible to infer which operations have occurred from a power trace of the device. Timing Analysis of program code can automatically identify when there is a divergence in the timing behaviour of object code. We have demonstrated this technique as a bug detection method in code from a wearable computer. It could be applied to security software as an automated defence in a compiler, producing warning when secure code has differentiable timing behaviours.

Scheduling Recent work [17] has demonstrated that code in embedded systems can be statically scheduled to remove overhead at runtime. In this work a client thread is statically integrated with a host thread. This integration task requires the identification of clock cycles in the host that are idle, into which the client thread can be inserted. For the timing guarantees on the client to be met, the idle regions must have the same timing properties as the client code, such as a fixed periodicity. Timing Analysis could be adapted for this usage as the fine-grained timing schema that is produced could identify idle regions that match the required properties.

Currently the Timing Analysis is limited to a simple class of processor. In order to apply the technique to more general processors there are several processor features that would have to be integrated into the execution model.

- Pipelines
- Caches
- Superscaler dispatch

Each of these features is inherently stateful, and would require an expansion of the timing states under abstract interpretation. Further research would be required to determine if the size of fixpoint produced would remain tractable with this extra information. It remains to be seen if the information known about these runtime states is precise enough to yield a useful result from Timing Analysis.

8.2 Precision Effects

The second effect studied in this thesis is the precision required to manipulate values within programs. In comparison with the timing effect, methods of automatically determining precision have more potential for increasing the productivity of software development in embedded systems, but are much more speculative. The results from this study are much less conclusive than the work on timing.

One reason for this is the intrinsic hardness of the problem. As argued in Chapter 6 the problem of accurately determining the necessary precision for each variable in a program can be reduced to finding solutions of arbitrary congruences — a hard problem in number theory. The result that we present for analysis is shown to be marginally more precise than previous results, but only in specific cases. Attempting an analytical solution to the problem requires an exponential amount of state, and so an analytical approach is only tractable for very small problems.

The approach that we have taken partitions the problem into three separate stages. The interface for the programmer is a domain specific language that allows the problems to be specified in a simple expressive manner. The analysis of the precision is split into a separate stage. The final stage is an automatic generation of program code from a pair of interpreters. This segregation into three parts means that the visible interfaces for the programmer are isolated from the hard analysis problem. The solution that we have supplied to this analysis problem could be substituted for other work in the area, or can be supplemented by annotations from the programmer themselves.

The automatic generation of code shows an unusual use of the Futamura Projections to create the compiler. We argue that writing the two interpreters is a simpler task than writing the compiler. Simplifying this task removes a major obstacle from applying the domain specific language to other architectures. Interesting future work in this area would be to investigate if either of the interpreters can be automatically generated from a formal description of the domain language, or a formal description of the processor architecture.

Bibliography

[1] Tor Aamodt and Paul Chow. Embedded ISA support for enhanced floating-point to fixed-point ANSI-C compilation. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architectures, and synthesis for embedded systems*, pages 128–137. ACM Press, 2000.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[4] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[5] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS*, pages 52–66, 1996.

[6] Peter Altenbernd. The false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, 1996.

[7] Johann Blieberger, Thomas Fahringer, and Bernhard Scholz. Symbolic cache analysis for real-time systems. *Real-Time Systems*, 18(2/3):181–215, 2000.

[8] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(6):1265–1296, 1998.

[9] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. Bitvalue inference: Detecting and exploiting narrow bitwidth computations. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 969–979, London, UK, 2000. Springer-Verlag.

[10] T. Langhammer C. Herrmann. Automatic staging for image processing. Number MIP-0410. 2004.

[11] Roderick Chapman, Alan Burns, and Andy Wellings. Static worst-case timing analysis of ada. *Ada Lett.*, XIV(5):88–91, 1994.

[12] Radim Cmar, Luc Rijnders, Patrick Schaumont, Serge Vernalde, and Ivo Bolsens. A methodology and design environment for DSP ASIC fixed-point refinement. In *DATE*, pages 271–, 1999.

[13] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2-3):249–274, 2000.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[15] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[17] A. G. Dean and J. P. Shen. Techniques for software thread integration in real-time embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, page 322. IEEE Computer Society, 1998.

[18] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG '00: Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 9–27, London, UK, 2000. Springer-Verlag.

[19] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

[20] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[21] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[22] Altaf Abdul Gaffar, Wayne Luk, Peter Y. K. Cheung, and Nabeel Shirazi. Customising floating-point designs. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 315, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. Unifying bit-width optimisation for fixed-point and floating-point designs. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *ICFP*, pages 74–85, 2001.

[25] Mohinder S. Grewal, Lawrence R. Weill, and Angus P. Andrews. *Global Positioning Systems, Inertial Navigation, and Integration*. Wiley, Inc., 2001.

[26] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[27] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.

[28] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[29] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 288, Washington, DC, USA, 1995. IEEE Computer Society.

[30] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, New York, NY, USA, 1972. ACM Press.

[31] M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[32] Manuel V. Hermenegildo, Francisco Bueno, German Puebla, and Pedro Lopez. Program analysis, debugging, and optimization using the ciao system preprocessor. In *Proceedings of the 1999 international conference on Logic programming*, pages 52–66, Cambridge, MA, USA, 1999. Massachusetts Institute of Technology.

[33] P. Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 134, Washington, DC, USA, 1998. IEEE Computer Society.

[34] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[35] Keding, H. and Hürtgen, F. and Willems, M. and Coors, M. Transformation of Floating-Point into Fixed-Point Algorithms by Interpolation Applying a Statistical Approach. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Toronto, Sep. 1998.

[36] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.

[37] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems-II:Analog and Digital Signal Processing*, 47(9):840–848, September 2000.

[38] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 298, Washington, DC, USA, 1995. IEEE Computer Society.

[39] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 254, Washington, DC, USA, 1996. IEEE Computer Society.

[40] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *RTSS '98: Proceedings of*

the IEEE Real-Time Systems Symposium, page 334, Washington, DC, USA, 1998. IEEE Computer Society.

[41] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.

[42] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Syst.*, 17(2-3):183–207, 1999.

[43] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *RTCSA*, pages 255–262, 1999.

[44] S. P. Amarasinghe B. R. Murphy S.-W. Liao E. Bugnion M. W. Hall, J. M. Anderson and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, December 1996.

[45] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[46] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *CASES*, pages 270–276, 2002.

[47] Microchip. PIC16F84 data sheet. Technical Report DS35007B, Microchip Technology Inc, 2001.

[48] A Mok. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80. IEEE, May 1989.

[49] Vivek Nirkhe and William Pugh. Partial evaluation of high-level imperative programming languages, with applications in hard real-time systems. In *POPL*, pages 269–280, 1992.

[50] G. Ottoson and M. Sjisdin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of The ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 47–55, June 1997.

[51] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.

[52] Ian Piumarta. *Delayed Code Generation in a Smalltalk-80 Compiler*. PhD thesis, Department of Computer Science, University of Manchester, Oct 1992.

[53] A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. pages 510–584, 1986.

[54] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[55] Neil Pollard and David May. Using interval arithmetic to calculate data sizes for compilation to multimedia instruction sets. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 279–284, New York, NY, USA, 1998. ACM Press.

[56] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[57] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[58] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Program. Lang. Syst.*, 21(2):175–188, 1999.

[59] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.

[60] Cliff Randell, Paul Duff, Michael McCarthy, and Henk Muller. Headracer: A head mounted wearable computer with ultrasonic position sensing. In *Seventh International Conference on Ubiquitous Computing - Demonstration*, September 2005.

[61] Cliff Randell and Henk Muller. Context awareness by analysing accelerometer data. Technical Report CSTR-00-009, Department of Computer Science, University of Bristol, August 2000.

[62] Cliff Randell and Henk Muller. Low cost indoor positioning system. In Gregory D. Abowd, editor, *Ubicomp 2001: Ubiquitous Computing*, pages 42–48. Springer-Verlag, September 2001.

[63] Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.

[64] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the AMS*, 89:25–59, 1953.

[65] Yvonne Rogers, Ian Taylor, Danae Stanton, Claire O Malley, Greta Corke, Silvia Gabrielli, Mike Scaife, Eric Harris, Ted Phelps, Sara Price, Hilary Smith, Henk Muller, Cliff Randell, and Andrew Moss. Things arent what they seem to be: innovation through technology inspiration. In *Designing Interactive Systems 2002*, pages 1–11. ACM Press, June 2002.

[66] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27(2):185–235, 2005.

[67] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[68] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, January 1989.

[69] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

[70] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Redmond, WA, 1993.

[71] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[72] Robert Endre Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[73] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[74] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *IEEE Real-Time Systems Symposium*, pages 144–153, 1998.

[75] Edward O. Thorpe. The invention of the first wearable computer. In *Second International Symposium on Wearable Computers, 1998. Digest of Papers.*, pages 4–8, Oct 1998.

[76] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. LMS, Series 2*, 42:230–265, 1936-1937.

[77] Robert A. van Engelen and Kyle A. Gallivan. Tight non-linear loop timing estimation. In *Proceedings of the 2002 International Workshop on Innovative Architectures*, pages 21–26, January 2002.

[78] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. Parametric timing analysis. *SIGPLAN Not.*, 36(8):88–93, 2001.

[79] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, November 1995.

[80] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 192, Washington, DC, USA, 1997. IEEE Computer Society.

[81] Willems,M. and Bürsgens,V. and Meyr,H. FRIDGE: Floating-Point Programming of Fixed-Point Digital Signal Processors. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, pages 1000–1005, San Diego, Sep. 1997.

[82] Michael Wolfe. Flow graph anomalies: What's in a loop? Technical Report CS/E 92-012, 1991.