ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Results of Case Studies: First Cycle

Deliverable number:	D14
Workpackage:	Assessment (WP8)
Preparation date:	1 November 2004
Due date:	1 November 2004
Classification:	Public
Lead participant:	Univ. of Bristol
Partners contributed:	Tech. Univ. of Madrid (UPM), Univ. of Bristol, Univ. of Southampton, Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

In this deliverable we describe the experiences with the first case-studies. Deliverable D-7 presented a number of proposed case-studies in detail, in this deliverable we have worked out four case studies in more detail, and have an initial set of results.

The first case is the static strength reduction on floating point operations. Although this case was the most advanced in terms of pervasive systems, it turned out to be the most rewarding in terms of the use of state-of-the-art specialisation tools. We can perform fixed-point analysis using standard program analysis tools, which has simplified the process of fixed point analysis, and has allowed us to experiment with various algorithms for fixed point analysis. One algorithm, a Kalman filter, has been studied in detail.

A second case-study that we undertook related to the timing analysis. This case study yielded interesting results, but did not use off-the-shelf analysis tools. As such, it is an interesting test case to show the use of analysis techniques, rather than the analysis tools.

The third case-study is an emulator of the PIC micro-controller written in Prolog. This emulator can be specialised with respect to a given (legacy) PIC program and given inputs characteristic of environments such as regular patterns on communication channels. The residual code can then be analysed in order to discover properties of the PIC program.

The fourth case-study is a program for access right management. This is the user's ability to access resources, for example a printer that one is walking past, at the printer's discretion. It is a rule based program that is written as an interpreter of the access rules. We specialise the interpreter in order to reduce the overhead. We find that there is virtually no overhead left in this case.

The case-studies provide both a demonstration of gains that can be made in the pervasive domain, and a genuine test of the scalability of the analysis and specialisation tools, since the interpreters and their specialisations are definitely not toy problems, but contain in some cases thousands of lines of Prolog code, with high-arity predicates.

Contents

1 Introduction

Over the second year of the project we have tested analysis and specialisation techniques, and their respective tools. We have selected a number of case studies from the pervasive domain, and used existing techniques and tools to identify how program analysis and specialisation can aid in developing pervasive systems.

In particular, we have selected the following areas that we expect specialisation and analysis techniques to help achieve an improvement:

Memory foot-print. Pervasive systems are typically designed with little memory, in order to reduce both production costs and system size. When developing software for pervasive systems, one wants to reduce memory requirements; both in the code memory, which is related to program size and the number of initialised variables, and in the data memory, which is defined by the number of variables used and the size of any scratch-pad areas used in computation.

Power consumption. To be truly pervasive, systems must be powered by batteries, or some reusable supply. Amongst the sources of scavenged power are solar cells, and the harvesting of kinetic or magnetic energy from the environment of the device. Power efficiency prolongs battery life, or reduces the size of power scavenging units. In less pervasive contexts where power supply is virtually unlimited, for example in a car, low power will ultimately reduce fuel consumption. Power consumption can be reduced in various ways, including using fewer instructions to achieve a goal, using simpler types of instructions to achieve a goal, and maintaining less state during the pursuit of the goal. There are other ways of reducing power consumption that are outside the scope of this deliverable, in particular the reduction of power consumption on wireless networks.

Correctness. It is often hard to upgrade the software on a pervasive system once it is out in the field. Even though virtually all software will be stored in some form of non-volatile RAM or FLASH memory, it is often logistically difficult to upgrade the software embedded, for example, in a washing machine.

Deliverable D7 identified six case-studies, describing each in detail. Three of the case-studies from deliverable D7 were selected for this deliverable, plus one extra case-study that we think is appropriate. Each of these four case-studies show improvements in at least one of the three areas above.

The four case-studies described in this deliverable form the first round of experimentation on the project. After concluding our study of these case-studies we will move to a second round, the results of which will be described in Deliverable D25. We expect two of the case-studies described in this deliverable to be extended to the second round, and two or three new case-studies that required more mature tools to be developed. The second round of case-studies will involve experiments that are more challenging of the tools, but which aim to achieve improvements in these same target areas.

1.1 Method

We present the case-studies below, in Sections 2–5. We have broken each-study into several sections, which are organised as follows:

Problem description. This gives a short introduction to the case study, and explains the objectives of this particular case-study.

Background. The background consists of other attempts to solve this problem. The problems are often well recognised in literature. Often, using program analysis tools simplifies the approach to solving the problem.

Approach. The approach details which program analysis techniques and tools are required to deal with this problem. We detail how the techniques were applied in order to achieve our results. We also highlight applicable differences between these tools and techniques and the conventional approaches.

Results. The results can be quantitative (reduction of memory requirements by X%), or qualitative (the ability to spot certain types of errors). Depending on whether the case-study works on legacy code or attempts to generate new programs, there may be comparisons with existing code.

Future Work. Some of the case-studies have posed more questions then they provided answers. Some of the future work might require us to come back to the case-study in the second round

We give a brief overview of the case-studies in the next section, whereupon Table 1 in Section 1.3 shows how each case-study contributes to the overarching aims.

1.2 Case-studies

The first case-study is the *Precision Interpreter*, and is discussed in Section 2. In this section we study the mapping of floating point operations onto fixed point operations. The method that we use is that we create an interpreter for arithmetic expressions; the interpreter is then specialised to a particular program, and we plan to then specialise it further to code. We did not set out to invent new algorithms for mapping floating point arithmetic to fixed point arithmetic, but merely to create a framework that easily allows experimentation with various algorithms. Although this test case was the most advanced in terms of pervasive systems, it turned out to be the most rewarding in terms of the use of state-of-the-art specialisation tools. We have obtained preliminary results on this case, and will continue our research into this case for the final round of case studies.

The second case-study that we undertook related is the *Timing Analyser* in Section 3. In this study we develop a program that analyses legacy-code, which informs us about the timings of the piece of legacy code. The timing information can be inspected and tell us whether the legacy code is behaving correctly or not. This case-study yielded interesting results, but does not use off-the-shelf analysis tools. As such, it is a case that shows the use of analysis techniques, rather than the analysis tools.

The third case-study is the *PIC Emulator*, in Section 4. In this case we model the functionality of a PIC processor [40] as an emulator written in Prolog. The PIC emulator is specialised with respect to a given (legacy) PIC program and given inputs characteristic of environments such as regular patterns on communication channels. We then apply analysis techniques to the specialised emulator in an attempt to discover properties of the PIC program, such as constant or undefined register values and detection of dead code and other forms of redundancy. The specialisation process uses the LOGEN partial evaluator and Binding Time Analysis that is developed in other parts of the project, and general-purpose analysis tools are applied to the specialised program.

The fourth case-study is the *Access Right Verifier*, in Section 5. In this case study we investigate access rights management. This is the user's ability to access resources on a server, at the server's discretion. We investigate using an interpreter that evaluates the access rules, and then specialisation of the interpreter to reduce the overhead of this approach. We find that there is virtually no overhead left in this case.

1.3 Contributions

Each case study exercises a particular set of techniques and tools, and aims to achieve one or more of the goals listed before. In Table 1 we list the four case studies and their main contributions.

The PIC Emulator and Access Control Verifier case studies will address the reduction in memory size in detail. We expect that the PIC emulator can point out how legacy PIC programs can run in a smaller foot-print, and the Access Control Verifier is an example as to how a program

Case-study	Memory	Power	Correctness
Precision Interpreter		$\surd$	
Timing Analyser			$\surd$
PIC Emulator	$\surd$	$\surd$	$\surd$
Access Control Verifier	$\surd$	$\surd$	

Table 1: Contributions of each case-study.

that would seemingly need a large memory foot-print can run in a smaller amount of memory. To an extent the Precision Interpreter also deals with reduced memory foot-print, but we expect the gains to be marginal.

Reducing power consumption should be achieved by three of the case studies. The Precision Interpreter should show how to execute a program with few fixed point instructions, rather than to emulate floating point (for embedded devices that do not support floating point natively, such as the StrongARM, PIC, and many DSPs). The PIC emulator should be able to spot dead code in legacy applications, and the Access Control Verifier case study should demonstrate that a complex rule driven program can run in a small memory foot-print.

The Timing Analyser and the PIC emulator address correctness – in both cases of legacy code. The PIC emulator addresses correctness in a low level analysis of the encode, spotting read-before-write hazards, and write-after-write instances (which may not be incorrect, but just address dead code). The Timing Analyser assesses the timing characteristics of dead code.

The two case studies study legacy code (the Timing analyser and the PIC Emulator) are both aimed at the PIC16F84 micro-controller, but both are aimed to be retargettable with a minimum effort. The two other case-studies work to high level languages, the Precision Interpreter operates on a domain-specific language, whereas the Access Control Verifier is implemented in Prolog.

An important part of the case-studies effort is to practically use the techniques and tools that we develop, and identify their applicability in the pervasive domain. To this extent, we have tried and employed different tools and techniques for each of the case-studies. A summary of which tools were used in which of the case-studies is presented in Table 2. The tools marked with a “($\surd$)” have only been used for preliminary experimentation at present. All but one of the case-studies require abstract interpretation. Ciao is used in all case studies except for the Timing Analyser, which does not use any tools, it applies program analysis techniques. The specialisation tools are used in the other three case-studies. More details on our experiences with the tools are in Section 6.3.

Case-study	Abstr Int	Ciao	CiaoPP	LOGEN	SP
Precision Interpreter	√	√	(√)	(√)	
Timing Analyser	√				
PIC Emulator	√	√	√	(√)	√
Access Control Verifier		√		√	

Table 2: Deployment of tools and techniques.

1.4 Future case-studies

Not all case-studies defined in Deliverable 7 have been addressed in this first cycle. In the second cycle of case-studies, we intend to address abstract interpreation and specialisation of matrix operations in the Kalman Filter and analysis of the Stream Interpreter.

```
initialise state;
forever {
  obtain readings;
  modify state using readings;
  produce output from state;
}
```

Figure 1: Structure of many low level data processing applications.

2 Precision interpreter

In this section we present a program that is used to compile code for digital signal processing tasks. We show that we can develop a simple interpreter for this problem (rather than a compiler), and how we can specialise this interpreter, and finally generate PIC code from this interpreter.

The ambitiousness of this case-study means that in this deliverable we just present initial quantitative results. We show that the approach is feasible by producing a working implementation. We will conclude this case-study in the second round by producing qualitative results to show the effects of specialisation on performance for this domain.

2.1 Problem

Many programs on pervasive and wearable systems are based around *filtering* input data. For example, an accelerometer will read accelerations 100 times per second, and we need a program to from these accelerations decide when a person makes a step. Or an ultrasonic microphone may pick up chirps, and we need a program to translate the timings of those chirps into a location.

The code for all those algorithms follows a fairly well established path, shown in Figure 1. The code to modify the state is usually very much straight line code, but may have a few decisions in it. It is unusual for the code to contain any unbounded loops (other than the outermost forever loop), because the readings must usually be processed in real time. Many programs follow the above model. Examples include Kalman filters, Particle filters, Windowed averages, etc.

A simple example of a program of this nature is shown in Figure 2. It is a crude but highly effective program to identify a person's steps given acceleration data from an accelerometer. The program calculates average energy levels over the last few samples, by using a decaying average. One average is changed at low frequencies only, the other has a higher frequency cut-off (governed by the constants 0.95 and 0.80).

For the code in Figure 2 to be used effectively, it needs to be run on a micro-controller. This

```
double lowFrequency = 0;
double highFrequency = 0;
boolean madeStep = false;
forever {
  double m = acceleration();
  m = m * m;
  lowFrequency  = lowFrequency  * 0.99 + 0.01 * m;
  highFrequency = highFrequency * 0.80 + 0.20 * m;
  if (highFrequency > lowFrequency) {
    if (!madeStep) {
      madeStep = true;
      step_detected();
    }
  } else {
    madeStep = false;
  }
}
```

Figure 2: Program to detect steps.

need arises both from the lower power consumption of the micro-controller, and also from the smaller form factor. These issues would currently prevent a more general and powerful processor from being used in the application. We can implement the program on a micro-controller in several ways:

- We can use a C compiler for a PIC and translate the code above. The multiplications on double precision floating point numbers will be translated to a library call that implements floating point semantics, each floating point operation will require 1,000-2,000 instructions, or 7,000-14,000 instructions in total.

- We can hand translate the code to PIC assembly. We actually made this step, and in the process we realised that the floating point numbers can be implemented as very low precision numbers. Sixteen bits accuracy is sufficient for the two state variables (`lowFrequency` and `highFrequency`), given that the input precision (`m` is only eight bits). We also noticed that the range of the numbers is completely fixed, since `m` will always be in the range [-2..2]. This fixed range obviates the need for any computations on the exponent part of the floating point number. Finally, we observed that changing the constants slightly, radically simplifies the computation. Instead of the number 0.80 we use 0.75, instead of 0.20 use 0.25, instead of 0.99 use 0.9921875 and instead of 0.01 use 0.0078125; all these changes replace repeated fractions in bit patterns with a number of the form 2^{-k} or $1 - 2^{-k}$. Hence, multiplication by the last number simply consists of a "shift right" operation by seven bits.

This version of the program takes 40 instructions to execute, gaining a factor 175-350 over the C version (which reduces power consumption of our program by two orders of magnitude).

- The third way, and the subject of our case study is to write a compiler that can take the program above and that, given a small set of annotations will generate code that is close to the program that was hand-crafted in assembly.

The advantage of the last solution is that the programmer can experiment with precisions and change their code without great effort.

This problem is a well known problem in computer architecture and compiler design, mainly in the areas of multi-media code generation (the MMX instruction set of Intel), and DSP code generation. In general one is trying to map an algorithm that is used on *real* numbers (the mathematical set $\mathbf{R}$) onto an implementation that uses *fixed point* numbers (the set of numbers $\{k2^n | k \in \mathbf{Z}. -2^b \leq k < 2^b\}$ for some $n \in \mathbf{Z}$ and $b \in \mathbf{N}$). The general problem is hard and requires intimate analysis of the problem, but with sufficient annotations a reasonable mapping can often be made. In the example above, some of the observations made when hand crafting code in PIC assembly can be added to the code as annotations at source level.

The issue is to have a balance in the number of annotations required. For some operations it is very hard to statically analyse the precision, or to reason about precision, for example division. For other operations, such as addition and multiplication, it is easy to reason about the precision.

With this case study, it is our goal to find a way to experiment with different algorithms that implement this mapping — we are not in particular interested in inventing a new algorithm to perform the mapping.

2.2 Background

There has been extensive work in the literature about directly specifying the precision of variables. This work ranges from simple statistical approaches to finding the range of a variable, through to analytical approaches based on propagating the precision between operators. In the literature that we consider, the language used to write source programs is a variant of C. This is usually an extension of C to allow variable types to be declared with annotations. Sometimes the annotations directly specify the size of the variables and which bit-positions are used. In some cases a new type is introduced that looks like a float to the programmer but represents a fixed point number for the compiler. Most of the approaches are directed at DSP applications and so offer support only for fractional fixed point numbers, not for arbitrary bit-positions within the word, above or below the binary point.

2.2.1 FRIDGE: Fixed-point pRogrammIng DesiGn Environment

The FRIDGE project [54, 29] started in the late nineties, and specifically aimed at programming DSP style processors. The input language is a superset of ANSI-C, with a data type for fixed point arithmetic. They have a tool that translates programs written in ANSI C (with just floating point numbers) into Fixed-C (with fixed point numbers), then they have a tool that goes from Fixed-C back to ANSI-C, translating any fixed pint operations into integer operations, with shift operations that keep the binary point in the right place. So, in effect FRIDGE offers a ANSI-C source-to-source transformation

In particular, no assembly code is generated; it is up to the back-end of the C-compiler to generate efficient code for fixed point operations. Although this makes FRIDGE portable to any architecture, it will probably generate sub-optimal code. For example, if one wants to multiply two 32 bit numbers which both represent numbers in the range [-1..1], then one needs a multiplication that discards the low 32 bits of the answer, rather than the high 32 bits. In order to achieve this in C, one will have to cast the int to a 64-bit long, perform the multiplication and shift the result right by 32 places. The compiler will have to spot this sequence of instructions and generate the instruction that multiplies two numbers and keeps the high part.

Internally, FRIDGE uses a propagation of interval arithmetic over the program. FRIDGE is applicable to a large class of programs.

2.2.2 Fixed Point Refinement

The Ocapi project at IMEC aims to provide an environment for the design of reconfigurable systems. As part of that, they have designed a component that maps floating point numbers to fixed point numbers [12]. The method that is applied by Cmar et al. is highly interactive, aimed at implementing the computations in (reconfigurable) hardware. Their main concern is to deal with the quantisation errors that are introduced when the results of computations are stored in a small number of bits.

Cmar et al. have chosen to perform all their computations in high precision, and only quantise the number if the user explicitly asks for a quantisation, for example by assigning the number to a fixed point variable (signal). The user has very fine control over the operations, as they can choose between signed and unsigned numbers (we only deal with signed numbers, for simplicity), specify rounding modes and saturation modes, in addition to specifying the number of bits and the exponent that are required.

2.2.3 Autoscaler

At the Multimedia Systems Laboratory of Seoul National University, Korea, two projects aim to optimise and compile fixed point code [30, 32]. Fixed Point Optimisation Utility by Kim et al. Similar to the FRIDGE project, the AUTOSCALER is a source to source code translation, taking ANSI-C and producing ANSI-C. The difference is the method employed to get to the answer. The AUTOSCALER approach defines the code generation as an optimisation problem, the cost functions is defined in terms of the overhead of the generated code, and standard optimisation algorithms, such as simulated annealing and linear programming are used to generate an answer.

In order to generate ranges, the AUTOSCALER uses a statistical approach, generating ranges that are usually big enough (but maybe not always). The algorithm is executed with an input signal which is generated based on statistical properties of the expected input, and the range of each variable is monitored. The advantage of this approach is that it works on any calculation, linear or non-linear, but it requires a model of the input signal.

2.2.4 Emdedded ISA Support

In [1] Aamodt et al. 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 that are redundant because of inter-operand correlations.

2.2.5 R2D2 project

Menard et al [39] are investigating the automatic synthesis of components within a reconfigurable computing context. The components are designed using floating point operations and then refined into a finished product using fixed point operations. Generally applicable techniques are used to synthesise appropriate architecture and then generate code to execute upon these architectures. This technique falls into the area of *architecture aware compilation*. They use a SQNR metric as a precision constraint, then optimise with regard to execution speed, within that contraint. The analysis techqniue used is data-flow propagation on the necessary precision.

2.2.6 Comparison

There are two approaches to the problem within the literature. For the types of programs that are under consideration in this context, there are no irregular control flows. All loops are bounded statically at compile-time and hence code can be considered to be straight-line. Given this lack of variability in control-flow it is feasible to perform an exact analysis over the program to propagate dynamic ranges from variable to variable. This produces a precise solution for the translation, no overflow errors can occur in the translation because it is a conservative over-approximation. The other approach is to simulate the program and to test the ranges of variables over a range of inputs. This gives a statistical solution with a low probability of overflow occurring in the output program.

The simulation approach will be more efficient as it produces a tighter specification for the program variable ranges. This increased efficiency comes at the cost of a possible overflow at runtime, and the problem of generating appropriate test vectors that cover the program correctly for each input. The analytical approach gives guarantees of correctness (with respect to overflow), but the hull of variable ranges is more conservative.

Our approach is an analytical method that increases the precision of the analysis whilst maintaining the safety guarantees that a conservative approximation gives. It differs from the previous approaches to the problem by describing and analysing the algorithm under development in a high-level domain-specific-language. This language uses real numbers and gives precise guarantees of bit-accuracy - unlike an algorithm described using floating point arithmetic. This algorithm is arbitrary precision, and can automatically be synthesised to the required precision for a particular problem.

One major benefit of the language approach is fine-grained control over the possible operations that can be performed in source programs. This is not available when performing source-to-source transformations on a pre-defined language where a lot of effort will go into handling corner cases. One example of this is the lack of an explicit division operation in the language - scaling is performed by multiplication with representable fractions. This removes the difficulty of analysing division operation which can destroy the precision of an analysis result (the worst case for any non-trivial interval is an infinite width result).

2.3 Approach

Our approach has two main novel features. Most previous approaches have focused on the use of a compiler to solve the problem. This has been necessary because of the overhead that an interpreter would impose. We use an interpreter, this provides a large decrease in implementation effort and allows us to try many different techniques for the analysis. The other novel feature is using a specialiser to automatically compile code from the interpreter. This technique is more common in higher-level languages but has not been tried at this low-level before. The goal of our research is to realise the benefits of these features by producing a system that has both clarity and simplicity. We believe that the introduction of these qualities will greatly improve the development process in this area.

Interpreters offer clarity and simplicity because they are written in code with only local effect. A pure interpreter is decomposed into a structure that reflects the decomposition of the problem. The lack of global effects, and the hierarchical structure can be represented directly in a functional or logical language without any extra overhead. Although the solution to the problem contains no overhead, there is still a considerable overhead in mapping from the language to this program structure for each instruction. This is called the interpretive overhead and is one reason that a symbolic implementation like this would not be suitable for the resources of our target system. In order to produce code that is efficent enough to meet these constraints we must compile from our domain language into a target object form, removing the interpretative overhead. This compilation is performed automatically by the ASAP tool-set (Ciao) using the techniques that we describe in this report.

Previous research has focused on solving the annotation problem, and in generating code for fixed point operations. Our goal is not to invent a new technique for defining the precision of operations — this area has been thoroughly explored by others, such as the projects described earlier. Our goal is to change the method used to solve the problem. We will show that the tool-set is powerful enough to *automatically generate* a compiler for the problem, from an interpreter. As the interpreter is easier to write than the compiler, this results in a significant reduction of effort for the compiler designer. Interpreters are simple enough to allow more experimentation, and avoid part of the system being set in stone. This increase in flexibility will have direct benefits — as a side-effect of our research we have found a new method of assigning precision.

For our approach, two interpreters are written. One is an interpreter of the domain language, in this case the domain is precision specified code. The second interpreter is of the target instruction set. At present, we have only implemented the first interpreter; the second interpreter will be implemented as part of Task 8.2. For the target interpreter it is important that each instruction in the target Instruction Set Architecture (ISA) has a one-to-one mapping with a predicate in the interpreter. We perform compilation by specialising the domain interpreter with respect to the program. This interpreter is written in a subset of Prolog that maps onto the predicates for instructions in the target interpreter. This corresponds to writing an interpreter in the target ISA for the domain language - but is much easier. This simplicity comes from writing in an extension of the target language, where that extension includes normal Prolog control-flow and constructs. At specialisation-time we specialise the domain-interpreter onto the extension, which gives a syntactic isomorphism when the target ISA. For practical reasons, we have written the domain interpreter directly in the extension (as described above), but this can be seen as equivalent to two applications of the first Futamura projection.

In the rest of this section describing our approach we shall describe the work flow for the user and compare it to a conventional approach in Section 2.3.1. In Section 2.3.2 we will describe the target domain and provide a motivating example. Then in Section 2.3.3 we will describe the staging issues in writing a multi-level interpreter. Then we shall cover the individual stages in Sections 2.3.4, 2.3.5 and 2.3.6.

2.3.1 Design process for generated compiler

Easy and rapid development of the domain language is our goal. In Figure 3 we illustrate the process used to compile a program from the source domain into the target form. We have shown both the conventional approach, based on writing a compiler for the domain, and our approach, based on writing an interpreter and automatically generating the compiler. The problem of writing an entire compiler is that it involves a lot of work and it is very hard to verify. The conventional approach to constructing the domain language is therefore reduced to using an existing language and modifying it to support the features of the domain. We will compare this process to our own development, and note that the alternative is much harder and lengthier. In the case of deriving precision in filter codes the normal language chosen is C and the compiler that is modified is the SUIF compiler [55].

There are four forms that the program exists in, that are common to both approaches:

Domain Specific Language (DSL) This language is defined by the user to reflect the problem domain. In the conventional approach the DSL is C with annotations to describe the precision of a set of variables chosen by the user. Our approach is not limited by the semantics of an underlying language and any appropriate set of language constructs can be defined as the DSL.

DSL Representation In the conventional approach the program is translated to a modified SUIF parse-tree. This requires modification of the SUIF format to change the parser. Our parsed

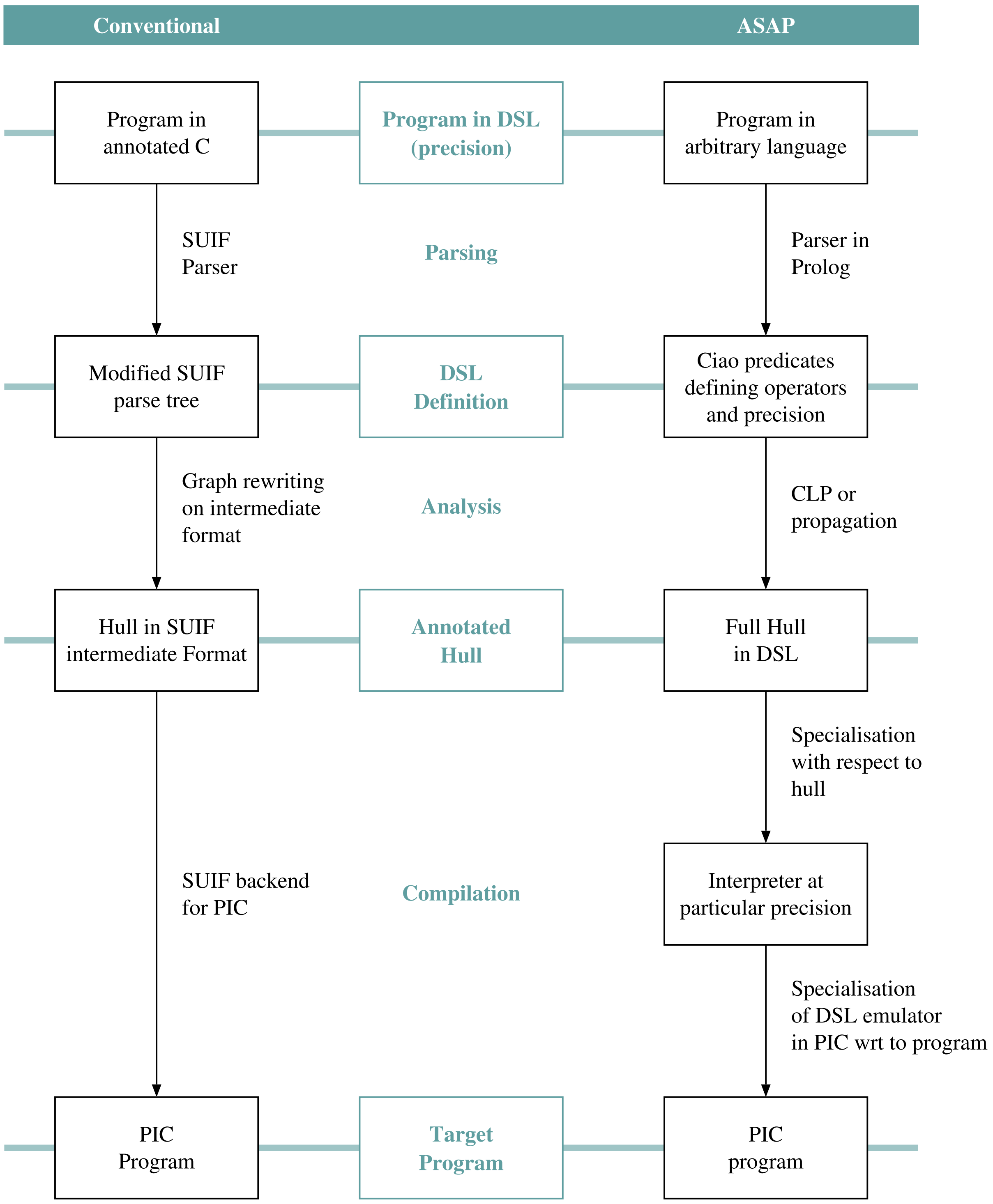

Conventional
ASAP
Program in annotated C
Program in DSL (precision)
Program in arbitrary language
SUIF Parser
Parsing
Parser in Prolog
Modified SUIF parse tree
DSL Definition
Ciao predicates defining operators and precision
Graph rewriting on intermediate format
Analysis
CLP or propagation
Hull in SUIF intermediate Format
Annotated Hull
Full Hull in DSL
Specialisation with respect to hull
Interpreter at particular precision
SUIF backend for PIC
Compilation
Specialisation of DSL emulator in PIC wrt to program
PIC Program
Target Program
PIC program

Figure 3: Comparison of approach to conventional approach

representation is as a set of Prolog predicates; describing each program point.

Annotated Hull At this stage, the set of program variables is fully annotated with the size of each variable. Various approaches are used in the literature to produce this full set from the supplied set of annotations. Two types of technique are statistical simulation and analysis. The former produces a tighter result with no guarantee of correctness. In contrast, the latter produces a safe over-approximation with no guarantee of efficiency.

Target In both the conventional approach and our own, the output is an object file in the ISA of the target processor. For this study our target is the PIC micro-controller. This architecture is not currently supported in SUIF (to the best of the authors knowledge). In the general case our goal is to target specialised micro-controllers that will not have support from mainstream compilers. We think that a fair comparison between approaches would include writing an appropriate back-end for SUIF.

Given these four forms, there are two processes that concern us. The process of developing this system (the work-flow of the domain designer) and the process of compiling programs from the source domain to the target language. The work-flow under the conventional approach requires the following tasks:

- Modify SUIF format / parser
- Perform hull analysis on internal SUIF representation
- Write a back-end for target system

The work-flow under our approach is much simpler for the designer.

- Write an interpreter for DSL in subset of Prolog
- Write an emulator of target system in Prolog
- Perform analysis on DSL

In the remainder of this section, we present the particular Domain Specific Language that we targeted, but it should be stressed that the method extends to other DSLs, *and to other algorithms for mapping reals onto fixed point representations*.

2.3.2 Domain Specific Language

The language that we use supports arithmetic, assignment, and a system-wide loop. Other operations can easily be added at the front-end, such as as conditional execution, bounded loops,or non recursive functions. However, the code above suffices to implement a Kalman-filter, the most complex operation we wish to investigate.

We assume that data is streamed into and out of our program. IE, each iteration of the system loop, some new data is available for processing. We assume that a surrounding system takes care of synchronisation and delivers the data to the loop. Similarly, we assume that output data is streamed out of our program.

The programming language supports two classes of variables, state variables (which hold their value between iterations of the loop), and temporary variables, which hold values inside the loop but that are reinitialised on every iteration. The state variables are updated exactly once on every iteration. The values that they are updated to are deferred until the end of the loop iteration, and they are changed between iterations of the loop (similar to single assignment languages).

The state variables must be annotated with their desired precision. This is necessary to ensure that we can construct a conservative approximation. Without a pre-determined 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. Only the algorithm designer can decide to what precision the state variables should be maintained. The temporary variables can be annotated if people wish to, and if no annotation is given, the precision interpreter will infer precision for those variables and all other (unnamed) intermediate results.

Apart from the state variables, the algorithm designer has to specify the precision of the input and output streams. Inputs on a pervasive system often come from hardware components, such as an A/D converter or a timer. All those components have a well defined precision. (Note that the precision of a A/D converter is hardware dependent, and that one may have to re-run the precision interpreter if the program is ported to a device with an A/D converter with different precision.) The designer decides what precision outputs they are interested in. For example, one may want to measure a location with 10cm resolution, as opposed to 2mm.

An example program is shown in Figure 4. This is a Kalman filter operating on one input and one state. It is easily extended to multiple inputs and multiple states, in which case one would define `x` to be a vector with each element having a specified precision. These precisions may be different because the elements of the vector can represent qualities in different units, or because our scale is different on each axis. We would assume that arbitrary expressions defining vector

```
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;

forever
{
  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 4: Kalman filter example code

indices were not possible, and hence any program using vectors could be translated to one in the scalar language that we work with. One interesting feature of the program is that the output statement is in the middle of the loop. This highlights the order independence of evaluation that occurs because of the semantics of the stateful variables. It is worth noting that introducing a temporary variable into the program (other than those that exist implicitly within the expressions) would remove this order independence, as temporary variables do not hold a dual state.

Once the program has been parsed it is represented as a set of predicates that describe the control data flow graph (CDFG). We show a graphical representation of this form in Figure 5.

2.3.3 Interpreter and staging

In this section we describe the interpreter of the DSL that the user of the system must construct. The interpreter must fulfil three functions;

Precision definition The predicates of the interpreter are an executable definition of the seman-

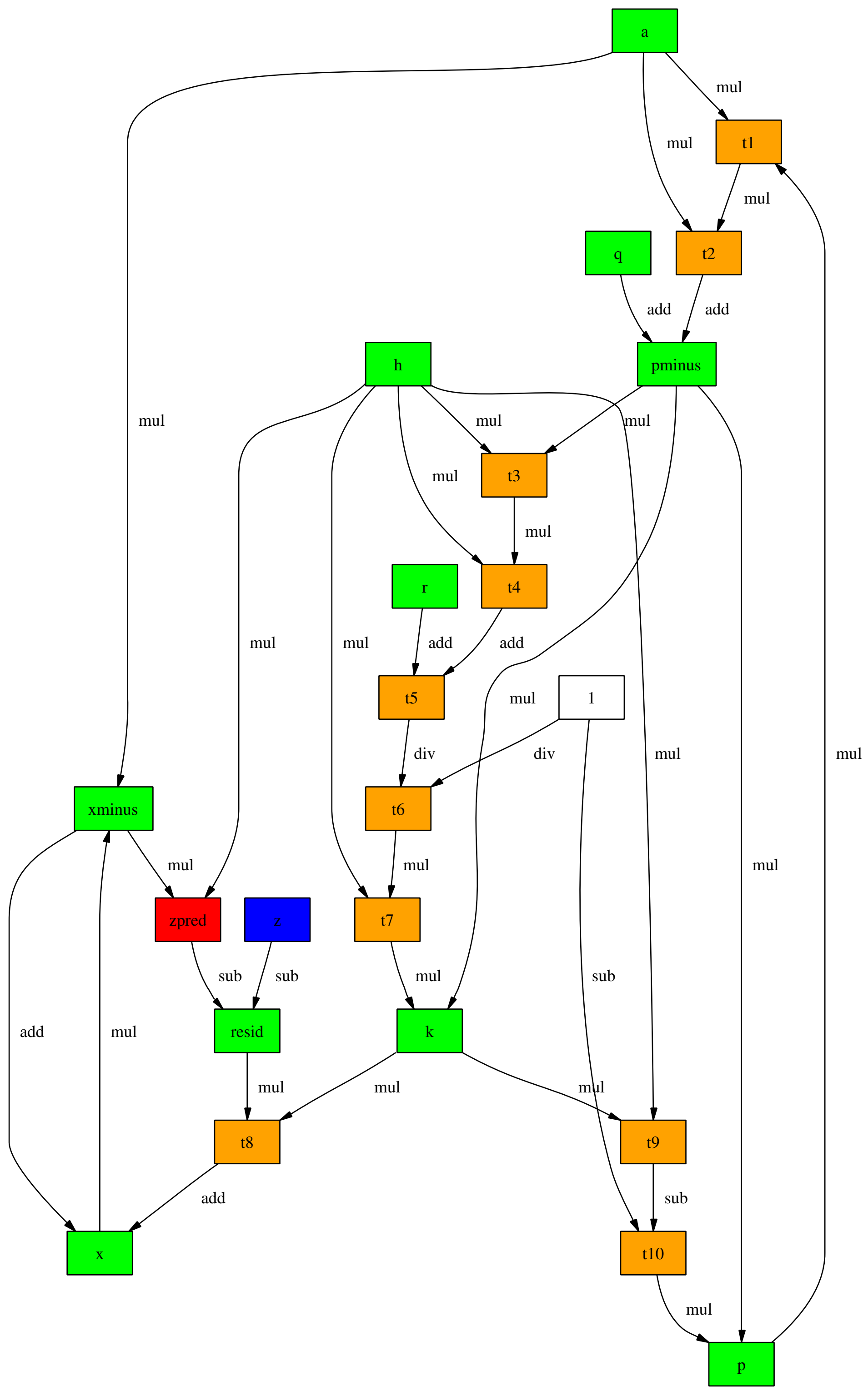


Figure 5: Control-Data Flow Graph for Kalman Filter

tics of each operator in the DSL, and how the precision of values is propagated through those operators.

Hull refinement If the current operation in the interpreter provides more information about the program hull than is currently known, then the hull must be refined with the extra information to produce a tighter analysis of the variable ranges in the program.

Operator definition Each operator in the DSL is implemented in the interpreter, thus it gives a concrete semantics for each operator.

As an example, consider an add operation in the DSL. Suppose one operand is known to be a number between -8 and 8 with a resolution of 0.25. We will represent this interval as $[-8, 8, 0.25)$. This interval is closed beneath (contains the value -8), and open above (does not contain the value 8). This operand can be represented in six-bits. The second operand is in the interval $[-2, 2, 1)$. The interpreter fulfil the three functions above as follows. The output precision of the operation is the interval $[-10, 9, 0.25)$. This is a seven-bit value. The semantics of the operations is that bit 0 of the second operand should be added to bit 2 of the second operand, and the other bits with the same relation. The final function of the interpreter is the hull refinement, this is the most difficult to encode. The seven-bit result for the add operation may be overly conservative, but in order to find this we need to propagate the precision that is actually used at a later point in the program back to this point. This is a global analysis problem (in the general case, although there are some specific cases that can be analysed locally), and it is very hard to encode this cleanly in a pure interpreter.

Our first experiment tried to perform the hull refinement within the interpreter. If this could be performed then all of the hull refinements are static computations that the specialiser will reduce out. This experiment was less than successful. The resulting interpreter was very complex. In general, an interpreter will handle state lookup and modification at the outermost level. The parts of the state being used for each function within the interpreter will be passed directly as values. This design creates an outermost loop isolating the complexity of maintaining the state, the rest of the interpreter can be seen as predicates modelling the operations in the language as relations between values. If we perform hull refinement as the interpreter is executing then we cannot maintain this clean separation between value mapping, and state modification. The precision of each value is passed into each predicate, and the resulting precision has to be integrated with the global store of precision ranges. This is unavoidable as the analysis required for hull annotation is a global problem. This annotation is encapsulated at each program point, and so we now have two global states to be manipulated. Unlike the value state, this precision state is bidirectional and so cannot be restricted to one outer loop.

The main result of the first experiment was that the resultant interpreter was too complex to perform an automatic BTA upon. The implicit mixture of global analysis code and local interpretation is beyond the current state-of-the-art. We did not consider it feasible to manually annotate thousands of lines of Prolog. This version of the interpreter failed to meet the research goals of simplicity and clarity. Instead we performed a second experiment, in which we wrote the interpreter as a multi-stage program. A stage is a partition of the computation based on the availability of data. Two normal stages within specialisation are compile-time and run-time. We have introduced a third stage; analysis-time. We separate the hull-annotation functionality from the semantic definitions within the interpreter and place them in a pre-phase. This stage of the code is executed before the specialisation of the interpreter. The result of this separation is that the entire precision hull is now a static parameter for the specialiser.

Staging is not a syntactic construct within Prolog. Examples of staging as language constructs are C++ templates and the language MetaML[52]. C++ templates are an example of embedding a compile-time computation within a language using staging. They are a syntactic construct within the language that has a defined stage (compile-time). MetaML offers syntactic support for the programmer to define arbitrary nesting of stages (delayed computations). Without this direct support in Prolog we have manually split the interpreter into two parts, and we then explicitly sequence them. A staging construct that was recognised by the tool-set (in particular as a hint to the BTA that one entire computation would conclude and produce static data) would allow this manual partitioning of the interpreter to be reversed, and give better localisation of the functionality. This approach is not feasible within the scope of this task, but could be considered in a later task (8.2).

Our manual staging of the interpreter has resulted in three phases. The initial input to the interpreter is the DSL program and its associated partial hull. This is provided to the analyser, which builds a complete hull. We describe the operation of this in Section 2.3.4. The output of the analyser (which can be seen as a stage within the interpreter that has been partitioned by hand) is a program with a complete precision specification for each variable. This program is executed by the domain interpreter. In Section 2.3.5 we describe how this interpreter is specialised to produce an executable program. In Section 2.3.6 we conclude this description of our approach by describing the generation of target code through the specialisation of the second interpreter.

Given a fully annotated program we then have an interpretative problem. For each point in the program there is enough local knowledge to execute the point. This problem has a natural expression as a program interpreter. We can then use the ASAP tools to specialise this interpreter. We are specialising with respect to the interpreter inputs: the program, and the ranges of each variable. The output of this specialisation is a Prolog program that operates on a stream of values. This residual program efficiently executes the target program at the specified precision. This result is still too heavy weight for our target architecture. For more heavyweight architectures (such as microprocessors) it would be feasible to compile this Prolog code into C, and then into a binary for the system. The high-level Prolog control-flows in the interpreter have been eliminated, leaving only the simple control-flows within the DSL which can be easily scheduled and converted to native code. The overhead of the interpreter, and of having an arbitrary precision algorithm are both reduced away.

2.3.4 Analysis Stage

The analysis stage can be seen be in two ways. In it an integral part of the interpreter, defining the semantics of the precision of values and how they propagate through the values in the language. But it is also a global analysis problem executed in a pre-phase. We have performed this separation manually, to create a stage that is an explicit module separate from the interpreter. Throughout this description it should be noted that with language support in the tools for explicit staging, this analyser could be an integral part of the interpreter, with the corresponding ease of use for the interpreter writer.

Representations The ease of use that we gain by developing the interpreter in a logic language can be seen when we define the analysis. In order to perform analysis of the values within the DSL we have to define a representation. This representation is isolated from the rest of the interpreter as we consider the output of this stage to be a static value. We have been able to easily experiment with two different representations for the analysis. Each representation defines the bounds of a variable, and the precision (or number of steps between values).

Interval Representation This domain uses a direct projection from the interval representation to the bounds. So an interval of $[-3, 4, 0.5)$ would allow the representation of the values -4,-3.5,-3 ... 3,3.5. Given the interval $[L, U, S)$ we can represent the value x where $L \leq x < U$, and $x_1, x_2 \in [L, U, S) : x_1 \neq x_2 \rightarrow |x_1 - x_2| \geq S$.

Logarithmic Representation This domain uses a representation of the exponents which are projected onto the value bounds using the base 2. So the same interval as above would be $[-2, 2, -1)$. Given the interval $[L, U, S)$ we can represent the value x where $2^L \leq x < 2^U$ and $x_1, x_2 \in [L, U, S) : x_1 \neq x_2 \rightarrow |x_1 - x_2| \geq 2^S$.

Given the representations of the operands for an operator in the DSL we can now define the representation of the resultant value. We now show the definitions for both representations, and show the result on a simple chain of addition statements in order to demonstrate the widening

Operation	R_U	R_L	R_S
R = A add B	$A_U + B_U$	$A_L + B_L$	min(A_S, B_S)
R = A sub B	max(A_U, B_U)– min(A_L, B_L)	min(A_L, B_L)– max(A_U, B_U)	min(A_S, B_S)
R = A mul B	$A_U \cdot B_U$	$A_L \cdot B_L$	$A_S \cdot B_S$
R = A div B	undefined	undefined	undefined

Table 3: Precision semantics of interval representation

that one representation imposes. The simple chain is shown in Equation 1.

$$x = y_1 + y_2 + y_3 + ... + y_n \tag{1}$$

Each operation that we consider is a binary operator, and so we use the same form for each $R = A \cdot B$ where A, B and R are all intervals with upper, lower and step elements, e.g. $A = [A_U, A_L, A_S)$. Firstly we consider the interval representation, which we show the precision semantics of in Table 3.

The result of the expression in Equation 1 then depends on the intervals of y. For simplicity we shall assume that all y have the same interval and that it is the simplest possible interval $[1, 0, 1)$, that is, each y is a single bit. We can now see that the resultant interval for x is $[n, 0, 1)$ and the bounds on x are $0 \leq x < n$. There has been no widening in the result.

If we consider the same expression in the logarithmic representation (Table 4) then we can see that for each of the n add expressions we will add at least 1 to the U of the sum so far. This will produce almost the same result $[n+1, 0, 1)$ but now this represents bounds of $2^0 \leq x < 2^{n+1}$. An exponential increase over the interval representation. This widening occurs because the worst case is that each of the sums overflows, but we can see that it is not possible for all of them to have sufficient value to overflow at each step. It is necessary to utilise some other information to restrict the intervals and produce a more efficient result.

Analysis techniques to compute the hull Previous research into the analysis has used the logarithmic representation. The advantage of choosing the logarithmic representation is that the operators in the language domain (addition, subtraction, multiplication, and division) all become linear in the analysis domain. Addition and subtraction map to successor and predecessor functions in the analysis domain. Whereas multiplication and division map to addition and subtraction. Analysis directly on the interval representation is difficult, as the propagation of multiplication operations is non-linear. It should be noted that all analysis operations on the logarithmic representation are linear.

Operation	R_U	R_L	R_S
R = A add B	$\max(A_U, B_U) + 1$	$\max(A_L, B_L)$	$\min(A_S, B_S)$
R = A sub B	$\max(A_U, B_U) + 1$	$\max(A_L, B_L)$	$\min(A_S, B_S)$
R = A mul B	$A_U + B_U$	$A_L + B_L$	$A_S + B_S$
R = A div B	undefined	undefined	undefined

Table 4: Precision semantics of logarithmic representation

The flexibility and ease that our approach gives us, allows us to experiment with different varieties of representation, and different techniques upon them. One possible solution to the difficulties that we have found is a *piecewise* analysis of the program. In areas of the code where the more accurate analysis can be performed, it can be used to propagate information to other program points. Where we are restricted in our choice of analysis we can use the other representation, and the techniques that apply to it. This combination produces a more accurate hull than either technique alone (as one widens results and the other is not applicable to the entire program).

Usage of CLP One analysis technique that we can easily experiment with is CLP. There is an interface to a CLP solver integrated into the tool-set. Our construction of the DSL means that all programs are forests of operations. Each tree is the expression for a single variable being assigned within the loop. The lack of dependencies between expressions means that we can treat each tree in the forest individually.

In each tree, all of the exterior nodes are either stateful variables, inputs, or outputs. This constraint is guaranteed by the language. (This guarantee holds in the case where all assignments in the language are to stateful variables or outputs. If arbitrary temporary variables are introduced then the number of unknowns increases, and this guarantee of correctness may no longer hold.) The root node of the tree must be either an output or a stateful variable. The leaves of the tree are variables referenced in the expression, and so must be either inputs or stateful variables. The precision of all of these types of variables is known by the analysis. All interior nodes are temporary variables which are to be annotated. As we have noted above, each operation produces three constraints in the form of linear equations. A range hull for the program must satisfy all of these equations. So we can formulate this as a CLP problem. We have a set of linear equations to satisfy, a set of known values, and a set of unknown values.

This CLP problem must always be satisfied. The number of unknown values is the number of temporary values. This is the number of interior nodes in the tree. The number of known values is the number of roots + leafs of the trees. These trees are binary. Hence there must always be more known than unknown values and the CLP problem is satisfiable.

We have investigated the use of a CLP solver upon the logarithmic representation successfully This experiment used the CLP(fd) module in Ciao, which is a CLP solver on finite domains. Direct specialisation of this code is not possible, and was one of the reasons that the analysis was moved to an earlier stage in the computation. We have also tried CLP(fd) and CLP(r) — a solver over the rationals — on the interval representation. We found that it does not produce a usable solution which is likely to be due to the non-linearity in the constraints.

Backwards propagation of precision The final analysis method that we have considered is backward propagation of precision constraints. The propagation that we have described in the forward direction in Tables 3 and 4 can be seen as the *produced* precision. An analogue would be to reverse the program and consider the precision that is *required* by each operation. For example, if the use of an add operation produced a result of 15 bits, but the use of that result only considered 6 of those bits, then we could more efficiently reduce the first computation.

The backwards analysis is also a global analysis problem, the interaction between the forwards and backwards analyses is that each reduction of the precision at a point in the hull requires that new range to be propagated via the other analysis until a fixed-point is reached. This analysis also requires an analysis stage in the computation prior to the interpreter. There is however a large number of solutions. In particular, it is often impossible to guess to which precision one needs to keep the answer of a multiplication or division, and when to round numbers. For example, consider the operation $A = (B+C+D+E)$ where B, C, D and E are all $[4, 0, 0)$ and A is $[6, 2, 0)$ in a logarithmic representation. In this case, we create two intermediate results, as in $X = B + C$, $Y = X + D$, $A = Y + E$. One can either perform all additions in full precision and then round the number in the final operation, or perform the roundings at an intermediate stage, when calculating X or Y.

Interestingly, because we are considering a program tree rather than graph, and because all of the precision inputs are known, it may be possible to specify the whole problem as an interpreter, and allow the specialiser to unroll the control flow for us. The backward analysis can be seen as a special case of the CLP problem where each part of the problem is locally over-specified, this allows information to be propagated using local knowledge. In the more general case the problem might be globally over-specified, but with local under-specification. This can be seen in any program that performs an operation on two variables which are being automatically annotated.

In some cases the entire program is locally over-specified. This is true of any program that we can reorder the operations in so that one of the two operands always has a known precision, in either direction. Note, this does not have to be the same operand. In these cases it is not necessary to converge to a fixed-point as there is a unique solution to the backwards analysis. We have found that the motivating example for our case-study — the Kalman Filter — can be written in this form, when applied to a single variable. In Figure 6 we show the results of the precision analysis on the example Kalman filter code using a backwards analysis. Each node in the graph shows a variable in the program. These are colour-coded according to their type, orange nodes are temporary variables, green nodes are state variables, red nodes are input streams and the blue nodes are output streams. Each node is annotated with the precision determined by the forward propagation, the backward propagation, and the hull that is obtained by taking the intersection of the two.

2.3.5 Specialisation of interpreter

Our approach uses four languages. We are constructing a DSL (*domain specific language*) which we will term D. There is a target language, in this case the *instruction set architecture* ISA of the PIC microcontroller, which we will term T. The tool-set is written in Prolog which forms our meta-language M. Lastly there is a restricted subset of the meta-language that we will write the interpreter in, we will term this M'.

The goal is to transform a program written in D (P_D) into T (P_T). In order to reach this goal, two interpreters are written. The first is an interpreter of D, written in T. We will term this interpreter int_T^D. This interpreter can be defined in the target language T, but it will be able to execute on the target system as it is very constrained in resources, and will lack both the program and data storage to implement the interpreter. We explain this point below. The second interpreter is an interpreter of the target instruction set architecture (ISA). This could be considered an emulator of the target system. This interpreter (int_M^T) is written in Prolog, and interprets the target ISA.

The key technique that we use, is a specialisation of one interpreter with respect to the other. We want to translate P_D into P_T only using the tools that we can construct, and as noted above int_T^D is not constructible. To perform this translation consider the specialisation of int_M^T with respect to int_T^D shown in Equation 2.

$$[\![spec_M]\!](int_M^T, int_T^D) \quad = \quad int_{M'}^D \tag{2}$$

The interpreter generated in Equation 2 is implemented in our meta-language Prolog, and executes programs in the DSL, but retains part of the structure of the target ISA. The structure of this interpreter will be specialised predicates, which map onto the instructions of the ISA that we want to target. This syntactic isomorphism means that the interpreter fulfils the property in Equation 3. For each program in D we can construct a translation of that program in T such that

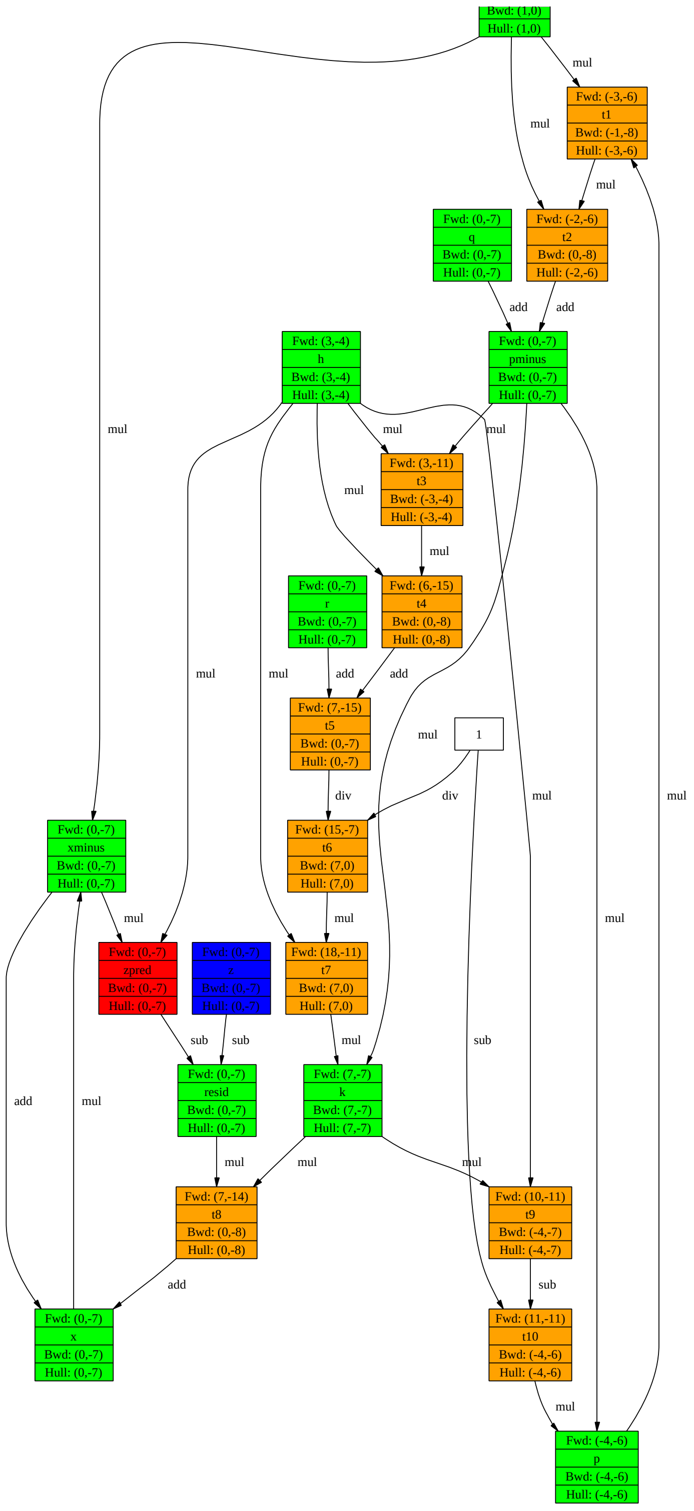


Figure 6: Precision Hull for Kalman Filter

it performs the same function, and has a syntactic isomorphism such that there is a one-to-one mapping between predicate calls and instructions in the target ISA (T).

$$\forall prog_D : \; prog_D(I) = prog_T(I) \quad \wedge \quad prog_D \leftrightarrow prog_T$$
$$\text{where } prog_T \text{ is } [\![spec_{M'}]\!](int^D_{M'}, prog_D) \tag{3}$$

The important aspect of this approach is that we can *directly* construct $int^D_{M'}$. For our experiment we have followed this approach, effectively performing the first specialisation by hand. This corresponds to writing the DSL interpreter in an extension of T. This extension is actually a subset of Prolog (M'), and so we can use Prolog control-flows and data representations, as long as they are specialised out of the final program. The other point of note, is that we have a tool to execute, and to specialise, M' directly *as it is a subset of Prolog* and hence we can use the normal tool-set.

This interpreter will be used during program development. We can execute the program under development on this interpreter, directly within the tool-set of the project as shown in Equation 4. For an improvement in performance we can also use the tool-set and the interpreter to compile the program into Prolog, as shown in Equation 5. This would allow the use of a Prolog-to-C compiler, and then a C compiler to automatically generate code for a targeted architecture. Whilst this technique is not efficient enough for our target micro-controller, it may be sufficient for targeting a workstation.

$$[\![P_D]\!](I) = [\![int^D_{M'}]\!]_M(P_D, I) \tag{4}$$
$$= [\![\; [\![spec^M_M]\!](int^D_{M'}, P_D) \;]\!](I) \tag{5}$$

In order to extend this work into a full compilation, from D to T, we need to further study the specialised program of Equation 5. At present we have merely experimented with a proof-of-concept. We need to run further experiments to scale this approach to the point where it can be performed upon our motivating example. This will be a demonstration of compiling a program that could not be done without specialisation — and in an easy and clearer manner than the conventional approach to compiler writing. We cover future work more fully in Section 2.5.

The use of three languages is quite a burden of knowledge, but we will note that the user of the generated compiler need only know D. Other systems require some knowledge of the underlying representation in order to debug programs. Our approach is based on source-to-source transformations and thus the intermediate representations are in forms of D. We assume that the constructor of the domain is familiar with M, or willing to learn — indeed the authors of the report learnt the meta-language during this project, and there is previous evidence [3] of the ease of developing interpreters in Prolog. This lowers the burden to achieving familiarity with the new source language D, and the target language T, which is true of any compiler.

2.3.6 Code Generation

The technical details of code generation have been covered in the previous section. Most of the technique used to ensure code generation is in the design of the interpreters that we specialise. In this section we present some of the details that are not related to specialisation.

The conversation from a specialised program into PIC code has been described as a syntactic isomorphism. There is a simple syntactic translation from predicate calls to instructions. All residual predicates contain straight-line code where each call is a predicate interpreting a single PIC instruction. This mirrors the syntax of PIC assembly language, and the predicates are simply rewritten without the surrounding Prolog syntax (e.g. commas between clauses).

There will be some post-processing necessary upon the PIC code, in particular we may need to perform register scheduling. In Section 2.5 we will describe methods to perform this work automatically. These methods will be the target of research in the second cycle of case-studies.

2.4 Results

Our approach is to provide functionality through a small *domain specific language* which is specialised for the target application. This approach contrasts with providing functionality through a library. Both approaches provide semantic checking of the usage of the functionality. In the case of the library this is performed syntactically through the definition of the API, and semantically through runtime checks and return codes, or exceptions to the control flow. In the case of a language we can perform both syntactic and semantic usage checks at compile time. We can also compile the functionality we are providing *with explicit knowledge of calling patterns and usage*. This extra information offers more optimisation potential than providing a static binary.

In this first cycle of case studies we have demonstrated the viability of constructing an interpreter that contains a syntactic isomorphism to the PIC instruction set. We have constructed a non-trivial example of a program in our chosen DSL, and have implemented it upon the interpreter. We have tested that our compilation approach works on a trivial proof-of-concept. These strong results show that we are capable of writing systems that could not be implemented directly because of the tight constraints of the target device. We have performed some preliminary tests with both LOGEN and CiaoPP, and we are positive that the tools can deal with the interpreter in its current form.

2.5 Future Work

In the second cycle we will continue to develop this case-study. Our first goal is to finish the code generation using the techniques that we have developed. This will demonstrate a qualitative success in the project by *automatically* generating a compiler for a domain that could not be implemented directly on the target device. The DSL that we have constructed allows arbitrary precision arithmetic to be specified by the user. This is not possible directly, as the microcontroller does not have enough resources to implement arbitrary precision code, and would be difficult to write by hand. This demonstrates a significant improvement in the development process.

We will then perform a qualitative comparison between this technique and hand-written code for the domain. We will compare the results of our compilation of simple arithmetic operations against a hand-written library provided by the vendor of the chip. We will consider the library to be written by a domain expert. This comparison will provide data on the performance of programs compiled through our automatic technique.

We can also compare the results with a standard C compiler for the domain. Once this definite goal has been achieved there are a number of interesting problems that we may pursue.

Sub-graph removal One intriguing line of research is suggested by the CDFG in Figure5. In this graph we can see that the bottom-left region is clearly a separate sub-graph. There are only control and data flows into this region from the rest of the filter, not out of this region. As this region contains both the input stream and the output stream this means that the rest of the filter is not dependent upon the run-time data, but only on the initial values of the state variables.

If the majority of the filter is not-dependent on dynamic data, but only on static data, then there are opportunities for specialisation to improve performance. There are two possible cases for the residual part of this computation. One case is that the data-flows from this sub-graph to the other sub-graph stabilise to a constant value. In this case, the entire graph could be replaced by a constant, giving a large increase in performance. The other case is that the values from this sub-graph form a cycle of a fixed-period. In this case, the computation could be replaced by a generator of that cycle. An increase in performance would result if the cost of executing the generating expression was lower than the cost of executing that part of the filter.

Executing this level of optimisation automatically though the tool-set may not be feasible. It would require the ability to identify and remove static computations many levels of interpretation deep, across the outermost loop in a deep nest.

Backwards analysis through specialisation Our work so far, suggests that it would be feasible to write a precision analysis that converged to result by applying the forward and backwards propagation rules in succession. An interesting area of study would be to use the specialiser upon this code to automatically generate a fast, efficient analyser.

Staging Prolog offers an extensible syntax and semantics. One interesting line of work would be to develop staging constructions and rewrite the analyser and interpreter as a single computation. The interesting research goal would then be to develop a BTA based on these stages.

Automatic determinisation During the development of the interpreter we had problems with there being too many choice points in the code for the tool-set to work with. This highlighted some bugs in the tools which have been fixed. One suggestion at the time was to use the determinacy analysis along with grounding patterns for the calls to each module to automatically insert green-cuts into the code and reduce the number of choice points.

3 Timing analyser

In this case study we investigate the techniques used to construct a tool that analyses the timing behaviour of pervasive systems. During the implementation of this tool we have not used the standard tool-set and techniques of ASAP. Rather, this case-study forms an initial theoretical study of the problem, and the tool implemented so far is a proof-of-concept. This case-study will be further developed in the second round of the project where we will apply the knowledge that we have learnt to implementing the tool on the Ciao Prolog system, and using the rest of the tool-set for the ASAP project.

3.1 Problem

In describing the class of systems applicable to our analysis, we use Pnueli's [43] categorisation of systems into two types; reactive systems that are non-terminating and must hold an ongoing dialogue with their environment, and transformational programs that take an initial input, execute for a (hopefully) finite amount of time and produce an output. Within this dichotomy we are concerned with reactive systems. We further divide this class into systems that are time-triggered — interacting with their environment at specific points in time — and those that are event-triggered — interacting with their environment at non-specific times but with specific types of interaction. Our analysis is applicable to time-triggered reactive systems.

The motivation for our analysis is a simple domain of reactive systems; sensor components from a wearable computer [47]. These sensors are composed of a microcontroller and an interface to a hardware sensor that samples the environment at precise moments in time. This combination of a programmable processor executing a well defined instruction set and an external interface with exactly defined timing characteristics is quite common in real time systems and so our analysis has wide applicability. Our concern is the verification of the timing behaviour of the components, not their logical functionality. In particular, we will note that in distributed systems where an embedded component has to communicate with other embedded components at precise times, we may consider the other components to be simpler sensors and actuators without a loss of generality. By considering interactions with other components to be of the same nature as interactions with sensors we can analyse any system composed of processor and sensor components within our domain of time-triggered reactive systems.

The systems that we analyse are repetitive, but not strictly periodic. Each is constructed from repetitions of a terminating procedure, but the procedure may not execute in a uniform amount of time, leading to a varying period for the system. Each statement within the program has a set of periods between successive executions. These periods are important because certain statements within the procedure cause side-effects that are externally visible. These statements are modifications and the reading of bits within control registers in the microcontroller. Setting bits in these registers drives pins on the microcontroller that control external hardware. Reading these bits polls the current state of external pins. By analysing the set of periods that each statement has, we are analysing the set of periods of these visible interactions. The set of all sets of statement periods forms a model of the programs timing behaviour that can be used to verify its temporal properties.

We are generating a model of the times at which each instruction in the program can be executed. This model consists of a set of timing configurations. Each configuration is a location within the program and a clock counter measuring discrete time since the program started execution. This model allows verification that the externally observable events within the program can only occur within a valid set of times.

For the program under analysis, we model the flow of control as a sequence of non-deterministic choices. In the instruction set that we are analysing all of these choices are binary. We will assume that either branch can be taken by the program. If we make this assumption for all branches in the program then the number of timing states is infinite. This follows from the observation that the looping branch can always be chosen, and that as each loop takes a positive non-zero number of cycles, new states are being generated for each iteration of the loop. The key transformation in our analysis is to partition the set of branches into those that form loops and those that do not. For looping branches we will only make a non-deterministic choice on the first execution. After they have been executed once we will deterministically choose the non-looping branch. As the program is of finite length, and each part can only be repeated a finite number of times, this ensures that the space of timing-states that we compute is finite.

It is only necessary to consider the first execution of a looping branch as a non-deterministic choice. This is because we are constructing the set of timing traces for all possible executions of that iteration. This set of traces therefore contains the possible executions of all iterations of the loop.

We have taken the problem of constructing the space of possible timing states and partitioned it into several smaller problems. We have the problem of deciding which branches in the control flow graph form cycles; this is decidable. We then construct a finite representation of the timing paths through this graph without executing each cycle more than once. In order to use these paths to construct the set of timing-states we would have to decide when each loop is taken, which is an undecidable problem. If we could decide analytically whether or not looping conditions were taken then we could decide if the program halted. This restriction means that our analysis is limited to programs only containing loop expressions that we can analyse. There is a wealth of material [25, 24, 53, 2] in the literature devoted to analysing the bounds of different type of loop expression, we shall not repeat this material here but simply assume that the programs we are interested in only contain loop expressions amenable to existing analyses.

This leaves the last problem of deciding which branches are taken within decisions. We contend that in order for the program to be correct, if there is a decision to be taken, then the timing behaviour of both branches must be correct otherwise the program can fail. If only one branch is meant to be taken then it should be a piece of straight-line code and not a logical decision. So the over-approximation that we make is one that a correct program should satisfy within the context of embedded systems.

The program under consideration may decide its actions on the basis of the information that is communicated from the sensors, but by considering the entire set of possible timings between observable events we may safely discard the process which selects one of these timings through some form of logical decision on the data contained within these communications. This set of possible timings is not as precise as one that also analyses the logic that the device performs, but it is a conservative over-approximation of the set. For verification purposes we wish to ensure that it is not possible for the device to operate outside of some specification of the times between observable events, thus the logical behaviour of the device is irrelevant for our purposes as the over-approximation of its behaviour must lie within the bounds defined by the specification.

3.2 Background

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.

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. This case is known as WCET (Worst Case Execution Time).

3.2.1 Major results in WCET

The initial results [31, 51, 41, 42, 45] in the field of WCET, follow this assumption of termination in order to guarantee a finite length worst case path. The restrictions placed on programs under analysis are that loop bounds are known a priori, there are no dynamic data-structures, unbounded recursion or dynamic references to functions.

WCET analysis is now a mature field [46] with the above model of program analysis refined to consider many architectural features such as RISC, pipelines and caches. Loop bounds can be automatically annotated in a wide range of cases. The precision of results has been increased through the elimination of infeasible code paths and a tighter mapping from source annotations to the underlying assembly language.

3.2.2 Differences from WCET

Our method offers an improvement over WCET analysis in several ways; WCET produces an upper bound on the execution of a piece of code - we are interesting in the exact set of possible times between events rather than an upper-bound upon them. This allows verification of the correct variances in times between events. This exact set is required in some scheduling problems that are encountered in embedded systems [16]. As WCET analysis considers the total execution time of a software component it is not capable of providing event to event timings across loop iteration boundaries within the same component. We are interested in the periodicities of non-terminating systems where-as WCET assumes that systems are terminating.

Our results are strictly more conservative than WCET in the context of how much of the potential state-space they consider to be reachable. They provide a more finely structured result in the context of how they represent that state-space which provides the answers to more precise questions such as event to event timings.

3.2.3 Applicability of WCET results to model generation

WCET analysis is split into three phases that perform different transformation upon the source program. The names of the three phases can vary between researchers although the purpose of each phase is generally the same:

Low-Level Analysis extracts the cycle accurate timing information from the low-level code representation, usually either machine code or an intermediate form within a compiler.

Flow Analysis is concerned with defining flow-facts for the program. These are infeasible paths which exist in the code but cannot be taken because of logical effects, variable bounds and hence loop bounds.

Calculation combines these two sets of information about the program into an estimate of the worst case execution time.

The analysis that we describe in this section is analogous to a low-level analysis, but based on a different set of assumption than those used in WCET. We are concerned with the extraction of cycle accurate path times from a program. Within the context of embedded systems these are sufficient to answer some useful questions about the system's simple temporal properties. In order to answer more complex questions about the temporal properties of reactive components we need to implement a similar flow analysis, and then construct equations to describe the periodic behaviour of the component. The extraction of flow facts can use existing source-level techniques from the field of WCET. The final generation of equations to describe the periodic behaviour is substantially different. The description of these two phases, as they apply to reactive components, is beyond the scope of this case study and will be the subject of future work.

3.3 Approach

Our approach in this case-study is an investigation of how to design such a timing-analyser for a specific architecture. This investigation was performed in Haskell using a standard compiler. The motivation behind this approach was to study both the theory and practice in a language familiar to the authors before rewriting the tool using the ASAP tool-set. The rewriting would increase the generality of the tool as the goal is to be able to specialise with respect to a given architecture. The tool that we describe within this section is tied to the PIC architecture.

3.3.1 Example code

The microcontroller that we use as an example is the PIC-16F84 [40]. This microcontroller is of interest because it offers a good ratio of MIPS to power usage, which makes it a common choice of controller within the wearable and robotics communities. The instruction set of the PIC gives a low code density as all operations have to be performed on a single working register rather than in any register as with RISC instruction sets. This simplicity gives the PIC ultra-low power consumption, using just 2 mW of power when active.

The code that we use as an example for our analysis performs the transmission of a byte on a serial interface. This code is shown in Figure 7. The serial interface is a simple teletype protocol with a start bit, 8 bits of data, and a stop bit. This simple protocol is useful for a system made up of distributed components as it doesn't require a synchronised clock at both end points. Instead, the start bit is used to synchronise the sender and receiver and then as long as the drift between the transmission and receiver clocks is within a small tolerance (5% of the transmission time) the data will be communicated correctly.

The sample code relies on fine-grained timing synchronisation to operate the serial line at the correct rate. The sample shown is for a serial line operating 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 we can manage on a discrete clock is every 22 cycles which is within the error tolerance.

To achieve this cycle accurate timing the sample code uses NOP instructions to pad the length of time required to iterate the main loop. 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 evidence of the trial and error required in order to produce the correct temporal execution.

3.3.2 Abstract Instruction Set

Each instruction within an ISA modifies the state of the processor. These changes in state can be broadly divided into three categories, and each instruction will then have effects in each of these categories:

Functional effects, such as modifying the contents of registers or flags.

Temporal effects which are the difference in the processors cycle counter from the beginning to the end of the instruction.

Control flow effects which modify the program counter (pc).

For the purposes of this analysis we are interested solely in the temporal and control effects that each instruction causes. By ignoring the functional effects of the instruction set we reduce the complexity of the analysis significantly, both in the technical sense that we reduce the size of the fixed-point that we compute, and also in the informal sense that it makes our analysis easier to understand and implement.

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 if it is set/unset. 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.

```
XMIT
    MOVWF  SER_TX
    MOVLW  NUMBIT+1
    MOVWF  BITCNT       ; Preset data bit counter

;   Send the start bit

    BCF    PORTB,TXD    ; Set start bit level
    GOTO   XMITC        ; Wait for start element
                        ; to go
;   Set the transmit data level from the carry and
;   wait for an element

XMITA RRF    SER_TX,1   ; Clock shift register RIGHT
                        ; through carry
    BTFSC  3,0          ; If data (carry) is ’0’,
                        ; skip
    GOTO   XMITB
    BCF    PORTB,TXD    ; Data is ’0’
    GOTO   XMITC
XMITB
    BSF    PORTB,TXD    ; Data is ’1’
XMITC
    ; call WAITEM       ; Wait for the element to go
    NOP
    ;NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP
    NOP

;   Count the elements as they are sent

    MOVF   BITCNT,1     ; Zero if just sent the stop
                        ; bit
    BTFSC  3,2          ; Skip next if bit count is
                        ; not zero
    RETURN              ; Exit from XMIT

    DECFSZ BITCNT,1     ; Dec. bit count, skip if zero
    GOTO   XMITA        ; Loop until all bits are sent

;   Bit count has zeroed, send the stop bit

    BSF PORTB,TXD       ; Set stop bit
    GOTO   XMITC        ; Wait for the stop bit to go
```

Figure 7: Sample serial transmission code

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

Figure 8: Abstract Instruction Semantics

When we ignore the functional effects of the ISA then these 35 instructions form 5 classes of equivalent control and temporal effects. Our analysis operates directly upon these equivalence classes which we term the abstract instruction set. For the PIC these equivalence classes are:

SINGLE Takes 1 cycle and pass control to the next address (pc+1)

RETURN Takes 2 cycles and return to the top address on the stack

SKIP Takes either 1 or two cycles and pass control to either pc+1 or pc+2

JUMP(n) Takes 2 cycles and pass control to the address n encoded within the instruction

CALL (n) Takes 2 cycles, passes control to address n and pushes the pc onto the stack

The operational semantics of these abstract instructions are shown in Figure 8. These transitions are between configurations composed of (a,b) where a is the program location and b is the number of clock-cycles since the program started execution.

In order to measure the length of program traces we must firstly construct a model of the timing characteristics of the ISA. In the case of the PIC micro-controller this model can be derived entirely from local effects within the code for the majority of the instruction set. Abstracting the functional characteristics of the instruction set we are left with the length of time to execute an instruction, and the locations that control is transferred to by the execution of that instruction. We do not consider the selection of which transition occurs in a given state which folds the instruction set into a simpler non-deterministic model. Each instruction in the abstract instruction set is a representation of a set of concrete instructions. Each transition is labelled with an integer number of clock-cycles, but there are no guards on the transitions.

The Call and Return instructions form transitions that are dependent upon a state (in this case the stack of return addresses). We do not represent this state within the timing states of the program. Instead we consider each locally reachable region of the program graph (subroutine) within an independent analysis and then use a call-graph to propagate information between them. We only cover the computation of a single reachable region within this case study. The reason for this is that to completely characterise the timing properties of a reachable region and then substitute it back into the calling region we need to use a flow analysis to deduce loop bounds and then generate equations that describe the complete timing behaviour of the region. This is beyond the scope of this case study. The consequence of this choice is that the analysis cannot determine the number of clock cycles that calls to subroutines will require and so is not suitable for code containing calls.

Conversion from PIC assembly code into the abstract instruction set is a simple many-to-one mapping for each location. Locations and therefore code-lengths are preserved when the program is represented in the abstract instruction format. The mapping of the sample program into the abstract instruction set is shown in Figure 10.

3.3.3 Timing Configurations

The configurations within an abstract interpretation are compositions of the state and the location that the state is valid at. In our analysis these states are the temporal state. The concrete temporal state is an integer, representing the number of clock cycles since program instantiation. The set of configurations with a common location then give the times that a statement can be executed at. This is potentially infinite for a repeating component.

We abstract these temporal states by using a relative clock. Instead of the number of clock cycles since the program started execution we use the number of clock cycles since the program passed a recording point. This gives a state that is a tuple of two integers; (a, b) where a is the number of clock cycles and b is the location of the recording point.

If we chose our recording points arbitrarily then we will not reduce the cardinality of the state space within the analysis. If there is a loop between the recording point and the current point of execution then arbitrarily large time values can be recorded by continuing to chose the branch that iterates the loop. Our solution is to make the recording points the backward edges of loops within the program control flow graph. This ensures that *every* loop that could be taken between the recording point and the current point causes a new recording point to be set. Therefore it is no longer possible to construct arbitrarily large lengths of time in the state-space and the size of the state-space must be finite. This follows from the observation that there are only finitely many locations within the program, and the distance from a recording point to another location without iterating a loop is bounded by the length of the program. Intuitively, our abstraction folds all iterations of a loop onto the first iteration.

This is not a strict abstraction. We are folding the states associated with all iterations of every loop, in the program, onto the first iteration of the relevant loop. However we are also recording more information about the path taken to reach the current location than exists in the concrete

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

Figure 9: Abstract Instruction Semantics On Abstract Time

state, as each state contains the last loop back-edge traversed. This means that states that were equal in the concrete domain are now differentiated in the abstract domain. This increases the size of the fixed-point generated slightly but the extra information recorded is useful for the determination of flow-facts in a later separate analysis.

If we represent our program by a pair of functions P, and L, such that P maps locations to an element of the set of abstract instructions, and L decides whether a location is a loop back-edge then we can define a parameterised transition system that gives the semantics of the abstract instructions when applied to abstract configurations. This is shown in Figure 9.

The SKIP instruction class is a set of instructions that tests logical decisions and passes control along one of two transitions. It implements a guarded choice of next instruction in the program that is used to build more complex control-flow structures such as guarded jumps. The non-determinism in our transition system is contained in the transition from the location of a Skip instruction to both the following instruction, and the instruction after it.

Given this definition of a transition system parameterised by a program function P and a program function, we can generate the configuration transition system for the timing of the program. We demonstrate this in Figure 10 which shows the P function for our sample program and the application of the parameterised transition system to this instance of P. This combination produces the transition system shown.

3.3.4 Computing Reachability Distance

The transition system that we construct for a program consists of tuples of the form (f, t, l). These give transitions in the program control flow graph and a distance in clock cycles of making these transitions. Given the set of locations that are loop-edges we can now compute the set of temporal configurations for the program.

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 location that control is being transferred from, f, is a loop back-edge we reset the recording point a' to a

$P(\text{Sample Program})$		Transitions applied to $P(\text{Sample Program})$
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,-1,0)
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 10: Creation of Temporal Transition System

((-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 11: Fixed-point for sample program

and reset the time to 0. Each new configuration is therefore $(t, c + l, a')$ when not reseting the recording point, and (t, l, a) when reseting 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

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 11. 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 that we wish to verify the temporal properties of, 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 back-edge 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)
```

Upon the fixpoint of the example program, the filtering expression (`filterSet cond fp`) yields the result $\{(25, 21, 25)\}$ which is 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, e.g. when polling sensors that transmit data encoded into the width of pulses modulated to some duty cycle.

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.

The analysis that we have presented can be seen as the first of three states in a full timing analyser. The second stage will detect 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 accurate expressions for the temporal behaviour of instructions within a program. We have not fully explained the integration of the call and return instructions in this work as in order to propagate timing information through the program call-graph this complete description of timing behaviour is required. The problem of fully analysing calls and returns in source programs will be the focus of future work. This future work will integrate all three stages described above.

3.5 Future Work

We have a working timing analyser written in Haskell which can be applied to programs for the PIC micro-controller. If this case-study is extended in the second cycle then the goal will be to generalise the work to arbitrary architectures. There are two methods that could achieve this.

Low Risk The simpler approach is to translate the timing analyser into Prolog, and design a data-structure defining the timing characteristics of the target architecture. The timing analyser could be specialised with respect to this structure, generating a timing analyser for a specific architecture.

High Risk The more interesting approach is to write a timing analyser that deduces the timing behaviour of an architecture by generating and executing programs on an emulator. This analyser could then be specialised with respect to a specific emulator, all of the generated programs would be static and thus the generation and testing would reduce to constant values representing the specific architecture.

4 PIC Emulator

In this section we describe a method of analysing embedded software. We have modelled the functionality of a PIC processor, commonly used in applications such as wearable computing, as an emulator written in Prolog. The PIC emulator can be specialised using one of the partial evaluator that we developed in the other workpackages. We specialise the emulator with respect to a given program and given input characteristics of the environment, such as regular patterns on communication channels. Analysis techniques can now be applied to the specialised emulator in an attempt to discover properties of the PIC program, such as constant or undefined register values, timing and synchronisation when connecting more than one PIC processor running concurrently and communicating - and detection of dead code and other forms of redundancy.

4.1 Problem

The problem that we are trying to address is to write a general tool that can be used to reason about programs that execute on simple embedded processors. We would like to establish whether an embedded program contains programming errors, can be executed faster, can execute using a smaller program, or can be executed using less dynamic memory.

Programming errors. Low level coding is an error prone process, and it is quite easy to accidentally use a memory cell before it is initialised.

Smaller program. It is not unusual for programs to be larger than necessary. Usually the program will link to a library with standard functions to, for example, perform input/output on an RS-232 channel, multiply two numbers, or write data to a USB bus. However, those operations are usually more general then required, and will contain code that will not be executed in the particular program.

Faster execution. Apart from removing dead code, it is not unusual that certain operations are redundant because their data values can be predicted by analysing the program statically. This will save execution time and hence processing power.

Fewer registers. Various parts of the program may use memory variables that are used only in a specific subroutine. For example, two or more subroutines may each use their own temporary variable, rather than share one temporary variable.

4.2 Background

A PIC processor is a small micro-controller used in, amongst others, wearable computer systems. It is small and has a low power consumption. The particular model number we emulate, the PIC16F84, has 35 single word instructions, 1024 words of program memory, 68 bytes of data memory, en eight-level deep hardware stack and two I/O ports.

The program memory of the PIC is flash memory that is programmed over two of the I/O pins of the PIC. When the PIC is powered up, it will start executing instructions starting at the first instruction in flash. PICs are extremely simple, in particular, they have no support for memory management, kernel-mode, etc.

Most approaches to this problem will attempt static dead code elimination, and register allocation. However, one of our interests is that we want to be able to analyse legacy code, rather than compiling new programs. In addition, the method that we have chosen is relatively processor independent.

4.3 Approach

The approach is to model the PIC processor as a Prolog program; in other words we construct an emulator for the processor in Prolog. The design of the emulator emphasises semantic clarity rather than efficiency. Once we have created an emulator, we apply a partial evaluator to the emulator with respect to a particular PIC program, producing a specialised version of the emulator. The key characteristic of the specialised emulator is that it has a systematic relation to the PIC program. Each program point in the specialised emulator corresponds to a program point in the PIC program. Hence the results of analysing the specialised emulator (a Prolog program) can be related directly to the PIC code. General purpose analysis tools based on abstract interpretation are then used to analyse the specialised emulator.

The processor is emulated as a state transition system. The state contains the values of registers, program counter, stack, accumulator, etc. Each machine instruction will, when executed on a given state, produce a new state. The emulator is given an initial state and a program to emulate. The emulator then works by executing machine instructions from the program one at a time, each time altering the state according to the current instruction.

The emulator is implemented in Prolog, and a predicate called `execute` contains the main loop that executes each instruction.

```
execute(Prog,State,Env) :-
      fetchInstruction(Prog,State,Instr,Arg1,Arg2),
      execInst(Instr,Arg1,Arg2,State,State1),
      simulateEnv(State1,StateOut,Env),
      execute(Prog,StateOut,Env).
```

A state is a grouping of lists and values of the form `state(Regs,PC,Acc,Stack)`. The environment can be used to simulate external input to the processor.

We implemented an `execInst` for each machine instruction in the PIC instruction set. As an example the `movwf` instruction that stores the value of the accumulator in a data register, is implemented as shown below:

```
execInst(movwf,Arg1,_,state(RegIn,PC,Acc,X),
                  state(RegOut,PCOut,Acc,X)) :-
      PCOut is PC+1,
      updateZeroBit(RegIn,RegTemp,Acc),
      updateData(RegTemp,RegOut,Arg1,Acc).
```

The implementation is not efficient but it is generic. The data stored in a state and the instruction set can easily be changed to emulate other processors or micro-controllers.

In the state information, the data registers are stored in a list of tuples. Each tuple is of the form `RegNr-(Vlist, RWlist)` where Vlist is a list of values that has been assigned to the register. The head of the list is the current value. RWlist is a list of `r(PCw,PCr)`'s and `w(PCw)`'s signifying that the register has been read or written. `PCw` is the value of the program counter at the instruction that wrote to the register, and similarly `PCr` is the value of the program counter a the instruction that read the register. For instance, if register 10 had been written a value, say 20 at instruction 35, that had again been read 3 times (at instruction 41,45 and 48) before it were overwritten at instruction 55, it would look like this `10-([30,20],[w(55),r(35,48),r(35,45),r(35,41),w(35)])`. We keep track of the access pattern of the individual registers to allow more detailed analysis of the PIC programs; this is discussed in detail Section 4.3.3.

The state of the processor is technically only the current values of the registers, but keeping a record of previous values and an access pattern, will allow for analysis of a wider range of properties. A few examples of properties that can be extracted from the register state information could be, locating a register that are read before a value is written to it and thereby reading an undefined value, locate a register that is written a value to but the value is not read again, locating registers containing a constant value and detecting at which program points a register contains live or dead values.

4.3.1 Interaction with environment

A global clock is part of the environment and can be used to e.g. synchronise the emulation of two processors connected to each other. The processor itself has no knowledge of a global clock, only an 8-bit timer. Registers 5 and 6 of the PIC processor are the processor's I/O ports. These registers form the processor's window to the environment.

Given a stream of data on the processor's input port (for example a stream of transitions from 0 to 1 and vice versa modelling a serial port having a specific data rate), the analysis can also tell whether there would be a read operation to the input port register in the time period that the data was available on the port.

Given a list of instructions P representing a PIC program, the emulator described in Section 4.2 is specialised with respect to a call to `execute` with first argument P, initial state values and program counter 0. Specialisation is performed using an on-line partial evaluator for Prolog, based on the approach described in [19]. The local control of the partial evaluator is tailored to handle the emulator, in the following way. All predicates are unfolded with the exception of `execute` and built-ins that are insufficiently instantiated. The effect of this is to produce a program containing only calls to versions of the `execute` predicate and arithmetic operations on the state. This effect could also be produced by an off-line specialiser such as LOGEN [33] by annotating the emulator program appropriately.

Furthermore, a different version of `execute` is generated for each call with a different program point (the fifth argument of `execute`) and set of registers that have been used. The effect of this is to produce at least one version of `execute` for each (reachable) program point. For example, the following two clauses both arise from executing an instruction at the same program point. `execute_32` represents execution of the instruction when registers $1, 3, 4, 5$ and 6 have been used. `execute_84` represents execution of the same instruction when register 16 has also been used. This approach generates more compact and precise representations of the state (only those registers actually used are shown) at the cost of producing larger specialised programs with possibly more than one version of each program point.

```
execute_32(B,[1-([0|C],D),3-([E|F],G),4-(H,I),
               5-(J,K),6-([L|M],N)],
               O, P, Q) :-
    R is E/4, S is Q+1, T is S mod 64,
    T==0, U is L+1,
    execute_33(B, [1-([1,0|C],[w,r,r|D]),
      3-([R,E|F],[w,r|G]),4-(H,I),
      5-(J,K),6-([U,L|M],[w,r|N])],
      O, 0, S).
execute_84(B, [1-([0|C],D),3-([E|F],G),
               4-(H,I),5-(J,K),6-([L|M],N),16-(O,P)],
               Q, R, S) :-
    T is E/4,U is S+1,V is U mod 64,
    V==0,W is L+1,
    execute_86(B, [1-([1,0|C],[w,r,r|D]),
      3-([T,E|F],[w,r|G]),4-(H,I),
      5-(J,K),6-([W,L|M],[w,r|N]),16-(O,P)],
      Q, 0, U).
```

The specialisation of `return` instructions needs special attention. When a subroutine is invoked by a `call` instruction, the program point following the `call` is pushed onto a stack. Upon exit from the subroutine by a `return` instruction, the stack is popped and control is returned to the program point at the top of the stack. There may be several calls to the same subroutine at different program points. Hence the next instruction to be executed after a `return` is not in general known at specialisation time. This can result in total loss of specialisation following a `return` instruction.

The problem was handled in a general way by regular approximation of the stack [23]. Here we obtain the desired result by pre-computing, for each `return` instruction in the program, the set of possible program points to which control might be returned. This is computable simply by searching the program for the `call` and corresponding `return` instructions. For instance, suppose that some subroutine is called at program points 108 and 113, and that the subroutine `return` instruction is at program point 58. Then we record a fact `returnpoint(58,[109,114])`, and similarly for all the other subroutines in the program. The emulator code for `return` is modified to enumerate the possible program points, so that in this example the version of `execute` for program point 58 will contain two clauses, whose bodies call the procedures corresponding to program points 109 and 114 respectively. At run-time, the top-of-stack value when executing instruction 58 must be either 109 or 114 and the correct version is chosen accordingly.

4.3.2 Analysis

Having obtained a specialised version of the emulator for a given PIC program, we apply program analysis tools in order to check properties of interest. The tools are based on abstract interpretation [13] and incorporated in the Ciao Prolog pre-processor.

The aim of analysis is to arrive at an abstract description of the program state at each program point. Various possible abstractions of the register states are considered. These include (1) abstracting the list of values by a constraint or interval; (2) abstracting the list of read-write operations by a regular description; (3) abstracting the list of read-write operations by one of a finite number of possible kinds, defined by types. These abstractions are discussed in more detail below.

A particular analysis consists of an *abstraction mapping* and an *abstract execution*. The abstraction mapping is a systematic transformation (based on the chosen state abstraction) applied to the specialised emulator, so that the transformed program operates on abstract states rather than concrete states. Abstract execution returns the model of the abstracted program (usually a fixpoint computation).

Abstraction mapping: Let us first consider the registers, which are represented as a list of terms of the form `Number-(VList,RWList)` where `Number` is the number of the register, `VList` is the sequence of values stored in that register and `RWList` is the history of read and write operations (represented as `r(PCw,PCr)` and `w(PCw)` respectively). In a concrete program execution `VList` and `RWList` might be unbounded in length since a register can be accessed an unbounded number of times.

To enable analysis of the specialised emulator it is necessary to abstract some of the information in the register list, so that it has a finite model. And to do this, a concrete regular type definition for the registers can be specified. One example of such a type definition is shown below:

$$\begin{array}{rcl} RegList & = & []; [RegInfo|RegList] \\ RegInfo & = & Number - (VList, RWList) \\ VList^{\alpha} & = & []; [Number|VList] \\ RWList^{\alpha} & = & []; [RW|RWList] \\ RW & = & r(Number, Number); w(Number) \end{array}$$

In general it is the programmer's responsibility to identify the unbounded components. The unbounded components can be identified by hand, by annotating the corresponding types. In this case, the unbounded components are the $VList$ and the $RWList$, which are recursive. The $RegList$ might look unbounded but is in fact always bounded, since there is a fixed number of registers of the processor. The above type definition can then be annotated to identify $VList$ and $RWList$. We use the superscript α to indicate that a component is unbounded.

It is the infinite components that must be abstracted; so the structure down to the α-level must be maintained and the structure below the α-level is abstracted away. The abstraction is therefore a selective one; we make sure that just enough is abstracted to ensure finiteness of the analysis.

An outline of a step by step procedure to ensure finiteness looks like this:

1. Declare type of the State (the register list in this case).
2. Identify possible infinite components of the type.
3. Annotate the type; that is marking infinite components with α
4. Transform the program by introducing the abstraction below the α-level. The procedure for this is: Traverse all the arguments in the program, where the arguments are considered as a tree, from the root to the leaves. Replace any subterm τ whose type is annotated as α, by a fresh variable U and add $\tau = U$.

Example 1 *Suppose the register list at some program point looked like this:*

```
[3-([21,22],[r(33)|A]),4-([1,10|B],[w(30),w(30)|C])]
```

The possibly unbounded parts are substituted with a fresh variable, and the following is obtained:

```
[3-(U1,U2),4-(U3,U4)]
where
[21,22] = U1
[r(33)|A] = U2
[1,10|B] = U3
[w(30),w(30)|C] = U4
```

The left hand sides of the introduced equations are themselves broken down, until all function arguments on the left of equations are variables. For instance `U4` *would be abstracted the following way:*

```
[U6|U7] = U4
w(30) = U6
[U8|C] = U7
w(30) = U8
```

The component ‘C’ is already a variable and it is therefore not replaced by another variable.

The abstraction is then performed by replacing the ‘=’ with a new symbol ‘$\rightarrow$’, where ‘$\rightarrow$’ is defined by a pre-interpretation[22]. See Section 4.3.3 for an example of an abstraction of the $RW List$.

The process of Abstract execution is the topic of the next subsections; for instance `U4` could be assigned the abstract value *WriteOnly* - one of a finite number of possible types.

4.3.3 Abstraction of the Read-Write History

As mentioned above, each register is represented by a pair consisting of a sequence of values and the history of read-write operations.

We wish to check relevant properties of the structure of the read-write history. For instance, a history with a single write followed by any number of reads represents a constant register value. A history containing only writes indicates redundancy - the register value is written but never read, hence the instructions that write to that register can be omitted. A history in which the first operation is a read (assuming the register is a "normal" register, not a hardware register) indicates an error since the value that was read is undefined.

Two different approaches are being followed: computing an abstract interpretation over *non-deterministic finite tree automata* [20], and abstraction based on a pre-interpretation based on given types [22]. In the latter approach we model properties of interest (such as write-only, read-before-write, and so on) using regular type rules. A read-before-write list rbw, for example, is described by the following type rule: $rbw = [\mathtt{r}(\_,\_)]; [rw|rbw]$ where $rw = \mathtt{r}(\_,\_); \mathtt{w}(\_)$. A history denoted sw with a single write (which should be first operation) is defined by $sw = [\mathtt{w}(\_)]; [\mathtt{r}(\_,\_)|rlist]$ where $rlist = []; [\mathtt{r}(\_,\_)|rlist]$. An analysis domain that models precisely the defined properties can then be constructed.

4.3.4 Liveness properties of registers

Keeping track of which instructions write to a given register, and which instructions subsequently reads the written value, enables us to determine at which program points the written value is live (meaning the value in the register will be used later on) and we can determine where the written value is dead (the written value will not be used again before a new value is written to the register). Since the PIC processor only has a limited number of register to store data, wasteful use of registers can limit the functionality that can be implemented on the chip. Remapping of the registers might free some of these register for more important use.

In a series of read and write operations on a register, beginning with a write (at program point w) followed by a number of reads ending with a final read (at program point r), a register is live at a point p, if p is reachable from w (that is, the execution of a number of instructions in the program will bring you from w to p) and r is reachable from p. Both w and r are available at the head of the $RWList$, if the head is a read operation. And the reachability criteria can be determined by constructing a directed graph of the program, where there is a vertex for each instruction and and edge between instructions that can follow each other. A branch instruction would for instance have two outgoing edges, and an instruction that is called from more than one place in the program, would have more than one incoming edge. This graph can be constructed from the specialised emulator. Reachability is then a search operation on the graph.

Registers that are live at disjoint parts of the program, can then be remapped to free up registers.

4.3.5 Abstraction of the Value List

The value list for a register is unbounded in length, as an unbounded number of write operations are possible. The tools for abstracting values and lists of values include intervals and convex hulls [14, 7]. For example, the list `[34,45,21]` could be abstracted by the closed interval $[21, 45]$ or by the linear constraint X where $21 \leq X \leq 45$. For a single register the two abstractions are identical, but convex hulls are more expressive and could represent, for instance, a constraint holding between several registers.

4.3.6 Constant Propagation

Some cases of constant values of registers may be discovered by approximation of the read-write history, as discussed above (write-once history). However this does not cover the cases in which the same value is repeatedly written to the register. Approximation of the set of values in the value history using intervals or convex hulls can be used to detect further cases. Specifically, a register is constant if its history of values is described by the interval $[n, n]$ for some value n.

4.3.7 Timing

The state of the processor includes a clock, and number of clock ticks taken to execute each instruction is built in to the emulator. This allows the emulator to record the clock value at each program point. For some applications, it is required that execution of some instruction (such as placing a value on an output port) takes place at defined intervals. Approximation of the clock value using convex hulls can check for such regularities.

4.3.8 Code Specialisation

Opportunities for code specialisation arise in various ways. Unreachable code might simply be detected by the specialiser; no versions of `execute` are generated for unreachable program points. Such code can simply be eliminated. Other unreachable code can be detected by analysing the register values at branch points in the program. Constant values detected for values controlling branch instructions can lead to elimination of some code branches.

4.4 Results

We tested the analysis tool on an accelerometer program. The program contains 320 instructions. The read/write history detected no use of uninitialised registers, and no dead computations.

Using the flow analysis method described in [20] on the specialised program, we found 266 instructions that could actually be executed. The remaining instructions belongs to a set of subroutines that are never called in the program.

4.5 Future Work

The results so far indicate that simple properties such as those described above can be obtained by general purpose analysis methods applied to the specialised emulator.

Current and further work aims to improve the scalability of the methods, and to examine analyses based on communication between the PIC program and its environment as well as possibly other PIC programs, via its input and output ports. We also plan to apply the “backwards analysis” method [21] in order to detect sufficient conditions on the external environment to guarantee lack of certain run-time errors.

5 Access Control Verifier

In this case-study we investigate the use of ASAP tools on an existing Prolog system. By quantifying the improvement in efficiency that we can produce on this system, we gain an insight into scope of programs that we can target at pervasive devices. This case-study relates to the specialisation of a meta-interpreter, which is an approach that will have uses in other areas of the project. We show a number of performance measures for our implementation, and we discuss the implications of using this approach. In particular, for this example, we show virtually zero overhead.

5.1 Problem

The issue of controlling a user's ability to exercise access privileges (e.g., is this person allowed to use this colour printer, or allowed to send a fax) on a system's resources has long been an important issue in Computer Science. In recent years, there has been considerable interest in access control, both in research and in practice. All aspects of security have been given particular prominence with the advent of pervasive systems and the Web. In a number of surveys, security issues have been reported by enterprises as being of paramount concern when deciding policies on the publication of Web data, and the availability of Web resources [10], and we expect that pervasive systems will raise many more concerns. The advent of Bluetooth and 802.11 devices has meant that mobile devices can use static infrastructure (such as Printers, SMTP servers, or coffee machines), but only if they have the right to do so.

In recent years, a number of researchers have developed some sophisticated access control models in which access control requirements may be expressed by using rules that are employed to reason about authorised forms of access [26, 9, 6]. In these approaches, access to resources are expressed by using rules that define the conditions that must be satisfied in order for a permission, denial or authorisation to hold. Expressing access control policies by using rules is natural, and enables many implicit permissions, denials and authorisations to be expressed in a succinct manner. However, an important practical issue that arises with the rule-based approach to access control is the problem of efficiently evaluating access requests when access control requirements are implicitly specified. This is especially a problem if the device that performs access control is not very powerful, and if access control policies are complex.

In many approaches proposals are made for attempting to ensure that access requests are evaluated efficiently when access control requirements are specified implicitly [26, 9, 6]. Jajodia [26], and Bertino [9] describe *view materialisation approaches* for attempting to optimise access control checks. The motivation for the view materialisation approach is to make explicit the access control information that is implicitly defined in rule form. Making explicit the implicitly specified access control information, means that access requests can be evaluated by considering explicitly recorded facts rather than these facts having to be derived at query evaluation time. Unfortunately, view materialisation is not so efficient to use when large numbers of *parametric derivation rules* [8] are used to express access control requirements and when the specification of access control requirements changes dynamically e.g., when *user session information* [6] is used in the course of deciding whether an access request is authorised.

Rather than using view materialisation techniques, the approach described by Barker and Stuckey [6] enables access requests to be efficiently evaluated by utilising *constraint logic programming* techniques [38]. This approach makes use of specialised constraint solvers, rather than view materialisation techniques, for the efficient evaluation of access requests in situations where large numbers of parametric derivation rules (e.g., rules that express temporal constraints on user access) would be expensive to compute, and when changes to an access policy are performed dynamically as a consequence of a user's session management. Nevertheless, the potential optimisation of access requests by using program specialisation techniques is not considered in [6]. Furthermore, each of the approaches described in [26, 9, 6] assumes that access control is expressed with respect to coarse-grained data objects (e.g., files and directories), and that an answer to an access request on a data item is simply whether access is allowed or not. In contrast, work by Barker [4] has the significant computational attraction of exploiting request modification techniques to combine the decision on allowing access with the actual generation of authorised data that may be released to answer a user's access request. However, this last approach does not exploit specific access request optimisation methods.

In contrast to [26, 9, 6], we describe an approach to the problem of access request evaluation where large numbers of parametric derivation rules are required in order to specify access policy requirements; where fine-grained access to data items is required (e.g., access to atomic formulae); where the answer to a user access request generates the set of logical consequences that the user is permitted to see; and where access control information is to be exploited for performance gains.

In overview, we describe an access control checker that is implemented by using a meta-program that is written as a logic program. The meta-program takes as input an access control program and a database of facts to which access needs to be restricted. We use a database as an example because there is a large volume of literature available in that area[26, 9, 6, 5, 15, 50]. We stress that our approach is not specific to databases and works on other applications of rule-based access control.

The approach that we describe enables the access control program to be specialised, in order to reduce the amount of run-time information that needs to be considered when deciding whether an access request is authorised. In effect, the approach ensures that a minimal amount of information is considered at access request evaluation time. Specifically, the user session information that applies at the time of an access request is used with a form of access control program that is specialised by using the relatively static information explicitly specified in the access control program.

In practice, the rules defining an access control policy are not subject to frequent changes. As such, this relatively static information may be exploited for program specialisation. Moreover, an access control request is a request that is made by a specific (authenticated) user to perform a specific operation (i.e., read, write, execute, etc.) on a specific database item. Exploiting the information about a user's identity and the access privilege the user wishes to exercise on a database object can be exploited to specialise a program for access control and hence can be exploited for computational advantage.

Although meta-interpreters have previously been developed for efficient constraint checking on databases [35], to the best of our knowledge, no approach has yet been proposed in the literature for generating specialised access requests via a meta-interpreter that manipulates access requests, access control policies and databases as object level expressions, and that pre-compiles access checking for certain access requests. In this paper, we describe a technique to obtain a specialised access control checker that is more efficient to use than using a database and access control program directly because some of the propagation, simplification and evaluation process is pre-compiled.

We consider the use of *role-based access control (RBAC)* policies [6] for specifying authorised forms of access to database objects. In $RBAC$, the most fundamental notion is that of a role. A role is defined in terms of a job function in an organisation (e.g., a *doctor* role in a medical environment), and users and access privileges on objects are assigned to roles. Moreover, access privileges on objects (i.e., *permissions*) are assigned to roles (e.g., a doctor has the permission to change a patient's prescriptions). $RBAC$ policies have a number of well documented attractions [50], and are widely used in practice [17]. Although we restrict our attention to $RBAC$ policies in this paper, it should be noted that $RBAC$ is a more general form of access control model than the *discretionary access control* and *mandatory access control* approaches that predate $RBAC$ [15], and the approach that we describe can be used with more powerful access control methods than $RBAC$ (e.g., the *status-based access control* model [5]). It follows that our approach is widely applicable.

We represent an $RBAC$ policy by using a logic program. The use of logic programs for representing access control policies has been recognised in a number of recent works (see, for example, [26] and [6]). Logic programs enable access policies to be expressed by using high-level declarative languages for which formally well defined semantics and operational methods with attractive theoretical properties (e.g., termination) are known to exist.

In the rest of this case-study, we discuss some preliminary logic and partial evaluation background in Section 5.2. Our approach is covered in Section 5.3, which covers using an $RBAC$ model and the formulation of $RBAC$ policies by using logic programs, and then we describe the meta-program that we use for the evaluation of access requests on databases with respect to a formulation of an $RBAC$ policy. In Section 5.4 we present a set of performance measures for the implementation of our approach, and we discuss these results. Finally, in Section 5.5, some conclusions are drawn and we make suggestions for further work within the project.

5.2 Background

In this section, we briefly describe the basic notations, and a brief overview of partial evaluation using LOGEN.

5.2.1 Syntax and Semantics

The $RBAC$ model and the $RBAC$ policies that we describe in later sections are expressed in the language of (function-free) normal clause form logic ($Datalog^{\neg}$), with certain predicates in the alphabet Σ of the language having a fixed intended interpretation. As we only admit function-free clauses, the only terms of relevance to Σ will be constants and variables. Hereafter, we denote variables that appear in clauses by using symbols that appear in upper case (at least the first character), and constants will be denoted by lower case symbols.

A normal clause is a formula of the form:

$$C \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n \qquad (m \geq 0, n \geq 0).$$

The *head*, C, of the clause above is a single *atom*. The *body* of the clause (i.e., $A_1, \ldots, A_m$, $\neg B_1, \ldots, \neg B_n$) is a conjunction of literals. Each A_i literal ($1 \leq i \leq m$) is a *positive literal*; each $\neg B_j$ literal ($1 \leq j \leq n$) is a *negative literal*. In the case of a negative literal, the relevant type of negation is *negation as failure* [11]. A clause with an empty body is an *assertion* or a *fact*. A clause with a non-empty head and a non-empty body is a *rule*. A *relational database* is a set of facts; a normal *deductive database* is a set of normal clauses. The set of facts in a deductive database Δ is referred to as the *extensional* part of Δ, (the EDB of Δ), and the set of rules in Δ is referred to as the *intensional* part of Δ (the IDB of Δ).

In our representation of a database, a fact of the form $p(c_1, \ldots, c_n)$ (where each subscripted p is an arbitrary n-place predicate and $c_i, \forall i \in \{1, \ldots, n\}$, are constants) is represented as an atom of the following form:

$$fact(p(c_1, \ldots, c_n)).$$

A clause of the following form (where each subscripted p is an arbitrary n-place predicate and each subscripted t is a term)

$$p_1(t_1, \ldots, t_n) \leftarrow p_2(t_i, \ldots, t_j), \ldots, p_m(t_k, \ldots, t_l).$$

is represented in our databases by using an atom of the following form:

$$rule(p_1(t_1, \ldots, t_n), [p_2(t_i, \ldots, t_j), \ldots, p_m(t_k, \ldots, t_l)]).$$

The access control programs that we consider are always *locally stratified* (a realistic assumption for most practical policies) and hence have a unique *perfect model* [44]. Having a 2-valued model theoretic semantics is important for ensuring that authorised forms of access are unambiguously specified.

5.2.2 Partial Evaluation and the LOGEN System

Partial evaluation [28] is a source-to-source program transformation technique that specialises programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the known part of the input. The so-obtained transformed programs are less general than the original, but can be much more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

Partial evaluation is especially useful when applied to interpreters. In that setting, the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialised version of the interpreter, which is sometimes akin to a compiled version of the object program [18].

The LOGEN system [34] is a so-called *offline* partial evaluator for Prolog, i.e., specialisation is divided into two phases, as depicted in Figure 12:

- First a *binding-time analysis* (BTA for short) is performed which, given a program and an approximation of the input available for specialisation, approximates all values within the program and generates annotations that steer (or control) the specialisation process.
- A (simplified) *specialisation phase*, which is guided by the result of the BTA.

Because of the preliminary BTA, the specialisation process itself can be performed very efficiently, with predictable results. Also, as shown in [36], the LOGEN system is well suited to specialise interpreters, something that we will aim to exploit in our approach.

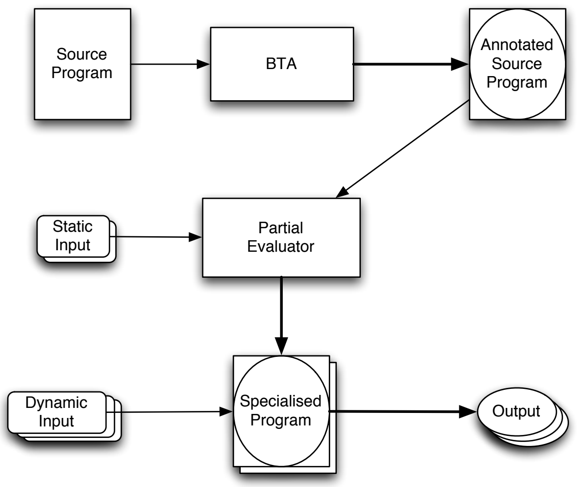


Figure 12: Offline Partial Evaluation

5.3 Approach

Our approach consists of implementing an access control checker as RBAC policies within a meta-interpreter, and using a specialiser to remove the interpretative overhead. This offers both the simplicity of a interpretative approach, and the efficiency of a direct access control method. We describe the representation of our programs in Section 5.3.1, and the meta-interpreter for these programs in Section 5.3.2.

5.3.1 RBAC Policies as Logic Programs

In this section, we describe a simple type of $RBAC$ policy that may be used to protect the information in databases. More specifically, the one type of policy that we describe here is based on the $RBAC_{H2A}^{P}$ model that is formally defined in [6]. We only consider one type of access control policy in this paper because our principal concern is to describe the generalities of using a meta-programming approach for access request checking, access policy program specialisation by LOGEN, and performance evaluation. It should be noted, however, that any of the policies from [6] may be represented by using our meta-programming approach to access control checking, with minor modifications.

We call an access control program that is defined in terms of the $RBAC_{H2A}^{P}$ model, an $RBAC_{H2A}^{P}$ *program*. This type of program is a finite set of normal clauses specified with respect to a domain of discourse that includes:

- A set $\mathcal{U}$ of *users*.

- A set $\mathcal{O}$ of *objects*.
- A set $\mathcal{A}$ of *access privileges*.
- A set $\mathcal{R}$ of *roles*.

In an $RBAC_{H2A}^{P}$ program, a user is specified as being assigned to a role by using definitions of a 2-place ura predicate, and the assignment of an access privilege on an object to a role is expressed by using definitions of a 3-place pra predicate in the $RBAC_{H2A}^{P}$ program. The semantics of these predicates in an arbitrary $RBAC_{H2A}^{P}$ program Π may be expressed thus:

- $\Pi \models ura(u, r)$ iff user $u \in \mathcal{U}$ is assigned to role $r \in \mathcal{R}$;
- $\Pi \models pra(a, o, r)$ iff the access privilege $a \in \mathcal{A}$ on object $o \in \mathcal{O}$ is assigned to the role $r \in \mathcal{R}$.

By separating the assignment of users to roles from the assignment of permissions to roles it is possible for user-role and permission-role assignments to be changed independently of each other in implementations of $RBAC_{H2A}^{P}$ policies. Thus, access policy maintenance is simplified (relative to the discretionary access control policies that were, until recently, used as a matter of course to help to protect the information in databases).

In the $RBAC_{H2A}^{P}$ model, specified in [6], an $RBAC_{H2A}^{P}$ *role hierarchy* is defined as a (partially) ordered set of roles. The ordering relation is a role seniority relation. In an $RBAC_{H2A}^{P}$ program Π, a 2-place predicate $senior\_to(r_i, r_j)$ is used to define the seniority ordering between pairs of roles. That is, the role $r_i \in \mathcal{R}$ is a more senior role (or more powerful role) than role $r_j \in \mathcal{R}$. If r_i is senior to r_j then any user assigned to the role r_i has at least the permissions that users assigned to rule r_j have. More formally, the semantics of the $senior\_to$ relation may be expressed thus:

- $\Pi \models senior\_to(r_i, r_j)$ iff the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC_{H2A}^{P}$ role hierarchy.

The $senior\_to$ relation may be defined as the reflexive-transitive closure of an irreflexive-intransitive binary relation ds. The semantics of ds may be expressed, in terms of an $RBAC_{H2A}^{P}$ program Π, thus:

- $\Pi \models ds(r_i, r_j)$ iff the role $r_i \in \mathcal{R}$ is senior to the role $r_j \in \mathcal{R}$ in an $RBAC_{H2A}^{P}$ role hierarchy defined in Π and $\neg\exists r_k \in \mathcal{R}\ [ds(r_k, r_j)\ \wedge\ ds(r_i, r_k)]$ where $r_k \neq r_i$ and $r_k \neq r_j$.

Assuming the lattice of role hierarchies to be complete, an $RBAC^P_{H2A}$ role hierarchy is defined by the following set of clauses (in which '_' is an anonymous variable):

$$\begin{array}{lcl} senior\_to(R1, R1) & \leftarrow & ds(R1, \_). \\ senior\_to(R1, R1) & \leftarrow & ds(\_, R1). \\ senior\_to(R1, R2) & \leftarrow & ds(R1, R2). \\ senior\_to(R1, R2) & \leftarrow & ds(R1, R3), senior\_to(R3, R2). \end{array}$$

In RBAC models generally, senior roles are assumed to inherit the access privileges on objects that are assigned to junior roles in an $RBAC^P_{H2A}$ role hierarchy. An $RBAC^P_{H2A}$ role hierarchy enables many authorisations to be implicitly defined, thus simplifying the expression of access control policies.

In RBAC, users activate and deactivate roles in the course of *session management* [6] as required to perform the tasks associated with a job function. In [6], the notion of a user $u_i \in \mathcal{U}$ activating a role $r_j \in \mathcal{R}$ in a session is represented by using a set of rules of the following form:

$$active(U, R) \quad \leftarrow \quad activate(U, R), C.$$

In this context, a user u_i requests to be active in a role r_j by appending an $activate(u_i, r_j)$ fact to an $RBAC^P_{H2A}$ program via a GUI. An $active(u_i, r_j)$ fact will be implicitly appended to an $RBAC^P_{H2A}$ program whenever u_i has requested to be active in a role r_j and the set of conditions C, on u_i's activation of the role r_j is satisfied. Any $activate$ assertion that enables the user u_i to be active in role r_j may be retracted by u_i when u_i no longer wishes to be active in r_j, and all of the $activate$ assertions for a user are automatically retracted when the user logs off of the system.

An ***authorisations clause*** [6] is used to define that a user $u_i \in \mathcal{U}$ has the $a_k \in \mathcal{A}$ access privilege on object $o_l \in \mathcal{O}$. In the case of $RBAC^P_{H2A}$ programs, the authorisations clause is defined thus:

$$\begin{array}{lcl} permitted(U, A, O) & \leftarrow & ura(U, R1), \\ & & active(U, R1), \\ & & senior\_to(R1, R2), \\ & & pra(A, O, R2). \end{array}$$

The rule that defines $permitted$ is used to express that a user U may exercise the A access privilege on object O if: U is assigned to the role $R1$, U is active in $R1$, $R1$ is senior to a role $R2$ in an $RBAC^P_{H2A}$ role hierarchy, and $R2$ has been assigned the A access privilege on O.

In the context of specialising an $RBAC^P_{H2A}$ program Π, we note that the definitions of ura, pra, $senior\_to$ and ds are part of the object level information that is used to protect the object level database in our approach. Moreover, the sets of clauses defining the extensions of the ura,

pra, ds and $senior\_to$ relations are static relative to the set of $active$ atoms that are implicit in Π. That is, the set of $active$ facts will change dynamically as users activate and deactivate roles. The aim of our approach is to specialise $RBAC^{P}_{H2A}$ programs to enable efficient access control checks to be performed by only considering user session information expressed via the set of $active$ facts that is current at the time of a user's access control request.

5.3.2 The Meta-interpreter

In this section, we describe the meta-interpreter that we propose for efficient access request evaluation on deductive databases that are protected by $RBAC^{P}_{H2A}$ programs. We restrict our attention to a consideration of read access.

The full Prolog code is part of the meta-interpreter that is used to execute the $RBAC^{P}_{H2A}$ programs that we have described for access control is shown in Figure 13. We use the following definition of `permitted`, as described in Section 5.3.1.

```
permitted(User,Op,Obj) :- ura(User,Role),
                          active(User,Role),
                          senior_to(Role,R2),
                          pra(R2,Op,Obj).
```

This paper only considers the definition of authorisations by `permitted/3`, as its goal is to apply $RBAC^{P}_{H2A}$ policies. Any number of alternative definitions of `permitted` may be used to implement different access policies (see [6] for other definitions of authorisation clauses that may be used), and do not require any modifications to our meta-interpreter in order to process those access requests.

Example 2 *Consider an $RBAC^{P}_{H2A}$ program Π with the following sets of facts:*

$$\begin{aligned} DS &= \{ds(r1, r2)\}. \\ ACTIVE &= \{active(u1, r1), active(u2, r2)\}. \\ URA &= \{ura(u1, r1), ura(u1, r2), ura(u2, r2)\}. \\ PRA &= \{pra(r1, read, s(\_)), pra(r2, read, p(\_)), \\ &\quad pra(r2, read, q(\_, \_)), pra(r1, read, r(\_, \_))\}. \end{aligned}$$

```
ura(steve,r1).
active(steve,r1).

pra(r53,read,p(_,_)).
pra(r53,read,cycle(_,_)).
pra(r53,read,tcp(_,_)).
pra(r53,read,q(_)).

:- table holds_read/2.

holds_read(User,not(Object)) :-
  \+(holds_read(User,Object)).
holds_read(_User,Object) :-
  built_in(Object).
holds_read(User,Object) :-
  permitted(User,read,Object),
  fact(Object),
  call(Object).
holds_read(User,Object) :-
  permitted(User,read,Object),
  derived(Object),
  holds_read_rule(User,Object).

holds_read_rule(User,Object) :-
  rule(Object,Body),
  l_holds_read(User,Body).

built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.
```

```
l_holds_read(_U,[]).
l_holds_read(U,[H|T]) :-
  holds_read(U,H),
  l_holds_read(U,T).

holds(U,O):- holds_read(U,O).

permitted(User,Op,Obj) :-
  ura(User,Role),  active(User,Role),
  seniorto(Role,R2), pra(R2,Op,Obj).

fact(p(_X,_Y)).
derived(cycle(_,_)).
derived(tcp(_,_)).
derived(q(_)).

rule(cycle(X1,X2),[p(X1,X2)]).
rule(cycle(X1,X2),[cycle(X1,X3),p(X3,X2)]).
rule(tcp(X1,X2),[p(X1,X2)]).
rule(tcp(X1,X2),[p(X1,X3),tcp(X3,X2)]).
rule(q(X),[p(X,Y),not(q(Y))]).

% For benchmarking query Q3:
b2 :- holds_read(steve,cycle(_,_)),fail.
b2.
bench :- ensure_loaded('database_cycle'),
    abolish_all_tables, cputime(T1),
    b2,
    cputime(T2), R is T2-T1, print(R),nl.
```

Figure 13: The full code of the access control interpreter, including a predicate `bench` used for benchmarking our query $Q3$. The predicates for queries $Q1$, $Q2$, and $Q4$ are very similar. The code above is intended for XSB Prolog, minor modifications were done for SICStus (e.g., replacing `cputime/1` by `statistics/2`).

Moreover, suppose that Π *is used to protect the following database* Δ *in which* p *and* s *are EDB predicates and* p *and* q *are IDB predicates:*

$$fact(p(X)).$$
$$fact(s(X)).$$
$$rule(q(X,Y),[p(X),p(Y)]).$$
$$rule(r(X,Y),[q(X,Y),s(X)]).$$

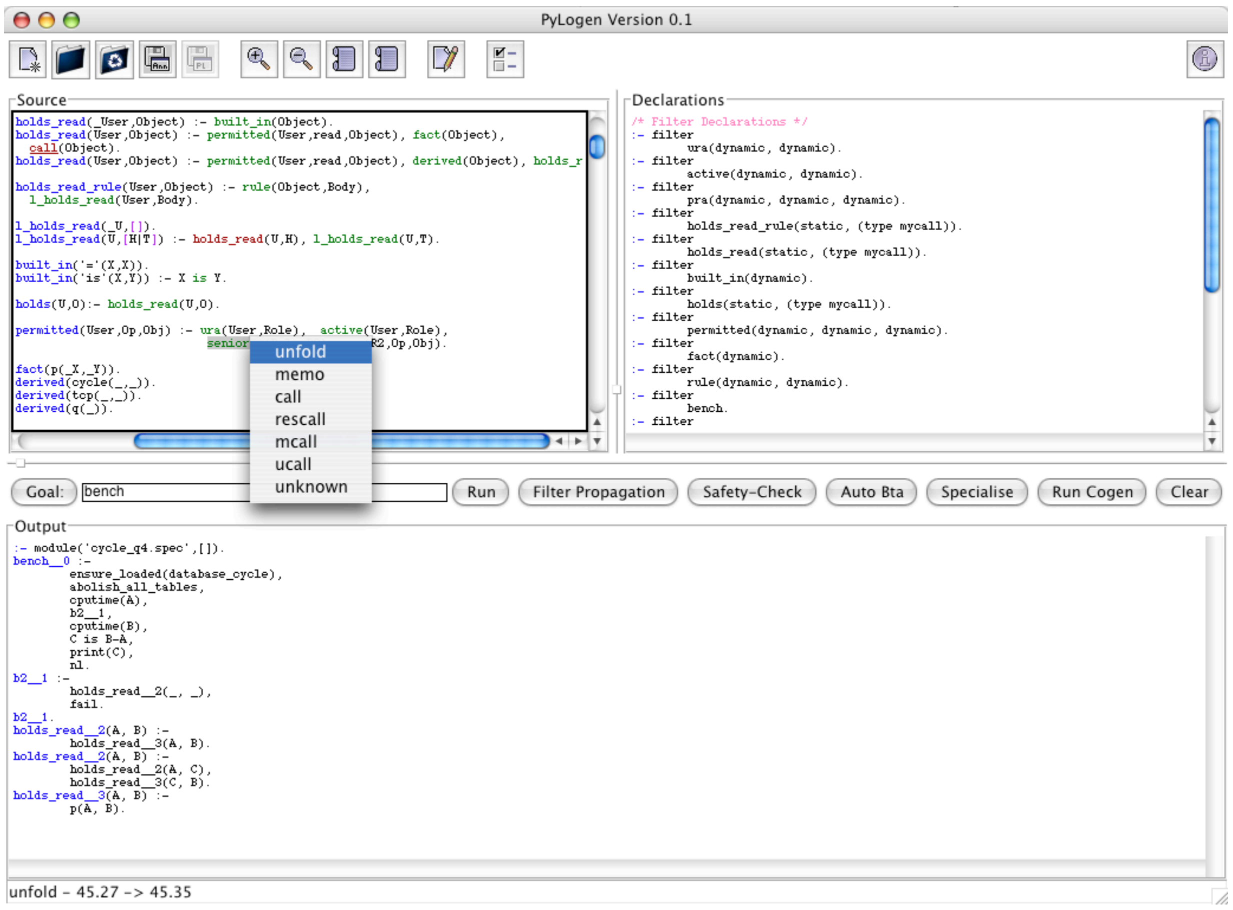
The access request `holds_read(u1,q(A,B))` *by user* u1 *to read all instances of* q *from* Δ*, can be specialised by* LOGEN *into:*

```
holds_read(u1,q(A,B)) :- holds_read__0(A,B).

permitted__1(B,C) :- active(u1,r1).
permitted__1(D,E) :- active(u1,r2).

permitted__4(B) :- active(u1,r1).
permitted__4(C) :- active(u1,r2).

holds_read__3(B) :- permitted__4(B), p(B).

holds_read__0(B,C) :- permitted__1(B,C),
                      holds_read__3(B),
                      holds_read__3(C).
```

By inspection, it is possible to see that the effect of such a specialisation is to reduce a predicate like `permitted`, which is defined in terms of the relatively static predicates ura, pra, ds and $senior\_to$, to tests on the run-time information that is generated in the course of session management, i.e., $active$ facts.

5.4 Results

In this section, we show some results for the meta-programming approach that we propose for evaluating access requests on deductive databases that are protected by using an $RBAC^P_{H2A}$ program. Our testing involved comparing the evaluation of access requests on (i) a non-specialised, and (ii) a LOGEN specialised $RBAC^P_{H2A}$ meta-programs. For comparison's sake, we also measured versions of the $RBAC^P_{H2A}$ that have no access control. These versions are implemented directly as Prolog clauses and hence needed no meta-interpreter to run.



```
holds_read(_User,Object) :- built_in(Object).
holds_read(User,Object) :- permitted(User,read,Object), fact(Object),
  call(Object).
holds_read(User,Object) :- permitted(User,read,Object), derived(Object), holds_r

holds_read_rule(User,Object) :- rule(Object,Body),
  l_holds_read(User,Body).

l_holds_read(_U,[]).
l_holds_read(U,[H|T]) :- holds_read(U,H), l_holds_read(U,T).

built_in('='(X,X)).
built_in('is'(X,Y)) :- X is Y.

holds(U,O):- holds_read(U,O).

permitted(User,Op,Obj) :- ura(User,Role),  active(User,Role),
                          senior                R2,Op,Obj).


fact(p(_X,_Y)).
derived(cycle(_,_)).
derived(tcp(_,_)).
derived(q(_)).
```

```
/* Filter Declarations */
:- filter
        ura(dynamic, dynamic).
:- filter
        active(dynamic, dynamic).
:- filter
        pra(dynamic, dynamic, dynamic).
:- filter
        holds_read_rule(static, (type mycall)).
:- filter
        holds_read(static, (type mycall)).
:- filter
        built_in(dynamic).
:- filter
        holds(static, (type mycall)).
:- filter
        permitted(dynamic, dynamic, dynamic).
:- filter
        fact(dynamic).
:- filter
        rule(dynamic, dynamic).
:- filter
        bench.
:- filter
```

```
:- module('cycle_q4.spec',[]).
bench__0 :-
        ensure_loaded(database_cycle),
        abolish_all_tables,
        cputime(A),
        b2__1,
        cputime(B),
        C is B-A,
        print(C),
        nl.
b2__1 :-
        holds_read__2(_, _),
        fail.
b2__1.
holds_read__2(A, B) :-
        holds_read__3(A, B).
holds_read__2(A, B) :-
        holds_read__2(A, C),
        holds_read__3(C, B).
holds_read__3(A, B) :-
        p(A, B).
```

```
unfold - 45.27 -> 45.35
```

Figure 14: Snapshot of a LOGEN session

The purpose of this section is more of an illustrative nature rather than actually presenting any theoretical nor empirical result. In this section we show the actual phase of specialising the $RBAC^P_{H2A}$ program by means of the LOGEN system. As shown in Figure 14, LOGEN is built with a Graphical User Interface (GUI) which facilitates the specialisation of logic programs. This snapshot illustrates how $RBAC^P_{H2A}$ program were annotated and specialised in the context of this framework. On the one hand, the *source code* of the database meta-interpreter (left window) is annotated by unrolling a list of options for each clause in a predicate. On the other, *filter declarations* are typed in the far right window, in order to guide the specialisation process. This allows LOGEN to, after just pressing a button, specialise the program for which the $RBAC^P_{H2A}$ policy has been optimised towards a particular query (i.e., the program is specialised according to the IDB of Δ).

The $RBAC^P_{H2A}$ programs that we use in our tests have included a definition of the $senior\_to$

relation that represents an $RBAC^P_{H2A}$ role hierarchy with 53 roles arranged as a complete lattice, and with each node/role of outdegree 3 or indegree 3. The $senior\_to$ relation has been materialised into a set of 312 pairs of ground binary assertions. In this case, we use a partial materialisation approach such that only the role hierarchy is materialised (but not the authorisations).

We have experimented with variants of the $RBAC^P_{H2A}$ role hierarchy by increasing the depth of the role lattice. The summation that follows describes the number of pairs of roles in the $senior\_to$ relation as defined by the $RBAC^P_{H2A}$ role hierarchy that we use in testing:

$$N + 2\sum_{i=1}^{d-1/2} 3^i + (N_d * P_{>1})$$

In the summation above, N is the total number of nodes in the role lattice, d is the depth of the lattice, N_d is the number of nodes at depth d in the lattice, and $P_{>1}$ is the number of paths of length 2 or greater from a node at depth d.

The unique bottom element in all of the $RBAC^P_{H2A}$ role hierarchies that we use in testing is assigned the read permission on all of the logical consequences of the databases that we use in testing. Moreover, our testing is based on a single user that is assigned to the most senior role/unique top element in the $RBAC^P_{H2A}$ role hierarchies/complete lattices that are used in our testing. Access requests are evaluated for this user. Our choice of user-role assignment and permission-role assignments imply that our tests are based on a worst-case scenario that involves the maximum amount of inheritance of permissions whenever an access request is evaluated.

The queries that we use in testing involve computing two binary relations tcp and $cycle$, and a unary relation q. The tcp relation is the transitive closure of a 2-place predicate p; the $cycle$ relation involves computing a transitive closure in order to determine elements in the reflexive closure of p; the definition of q is a variant of the well-known win program. The win program describes a two-player game in which a player wins if his or her opponent has no move to make. The formalisation of this two-person game may be expressed by the clause: $win(X) \leftarrow move(X,Y), \neg win(Y)$. The tcp program was chosen for inclusion in testing because of its practical significance; $cycle$ was chosen because it involves some expensive recursive processing; the q program was chosen because it combines recursion and negation, and is a useful benchmark test for performance studies.

The definitions of the tcp, $cycle$ and q predicates are expressed in our database thus:

$$\begin{array}{rcl} tcp(X,Y) & \leftarrow & p(X,Y). \\ tcp(X,Y) & \leftarrow & p(X,Z), tcp(Z,Y). \end{array}$$

$$\begin{array}{rcl} cycle(X,Y) & \leftarrow & p(X,Y). \\ cycle(X,Y) & \leftarrow & cycle(X,Z), p(Z,Y). \end{array}$$

$$q(X) \leftarrow p(X,Y), \neg q(Y).$$

The 2-place p predicate is defined by a set of 2495 facts. A total of 499 p facts are used to represent the chain:

$$p(a_1, a_2), p(a_2, a_3), \ldots, p(a_{498}, a_{499}), p(a_{499}, a_{500}).$$

An additional 1996 p facts are used to achieve a fan-out factor of 5 [48]. That is, for each p fact with the first argument a_i, where $1 \leq i \leq 499$, there are four p facts with the second argument of p equal to the value b_j, where $1 \leq j \leq 4$. For example, $p(a_1, b_1), p(a_1, b_2), p(a_1, b_3), p(a_1, b_4)$. For the $cycle$ program, at the n^{th} call to $cycle$, a chain of $(n-1)$ elements in the transitive closure of p is computed, and hence the goal clause $p(a_n, a_{n-1})$ is evaluated. An additional fact $p(a_{500}, a_1)$ is added to the 2495 p facts used with tcp to represent the end of the cycle.

The successful $tcp(a_1, a_{500})$ query that we use in our testing involves computing a 500 element chain starting from the element a_1 and ending with the element a_{500}. To evaluate the tcp query by using SLD-resolution, a search space comprising 499 SLD-trees with root $\leftarrow tcp(a_n, a_{500})$, where $1 \leq n \leq 499$, was generated. Each of these 499 SLD-trees spawns 5 subtrees; 4 of which fail, and one that succeeds. The four failing cases have a b_j value $(1 \leq j \leq 4)$ as the second argument of a p fact; the succeeding subtree terminates with an answer clause of the form $p(a_s, a_t)$ where $t = s + 1$, $1 \leq s \leq 499$ and $2 \leq t \leq 500$. The $cycle(a_1, a_1)$ query used involves computing every chain from a_1 to a_w $(2 \leq w \leq 500)$ in the transitive closure of p, until $p(a_{500}, a_1)$ succeeds and hence $p(a_1, a_1)$ succeeds. The failing query in our suite of tests ($tcp(a_1, a_{501})$) is an attempt to compute a 501 element chain that terminates at the element a_{501}. The successful $q(a_1)$ query involves generating 499 failing SLD-trees for the 499 evaluations of the $\neg q(c_m)$ subgoal, where $1 \leq m \leq 499$. The one successful SLD-derivation is generated from the ground clause: $q(a_1) \leftarrow r(a_1, c_{500}), \neg q(c_{500})$.

The results of the testing of our example queries are summarised in Table 5 (for the non-specialised case), and Tables 6 and 7 (for the specialised case). The queries denoted by Q_1, Q_2, Q_3 and Q_4 in these tables have the following meanings:

- Q_1 is the successful $tcp(a_1, a_{500})$ query run 10 times;
- Q_2 is the failed $tcp(a_1, a_{501})$ query run 5 times;

- Q_3 is the successful $cycle(X, Y)$ query (all 145,850 solutions);
- Q_4 is the successful $q(X)$ query (all 1,170 solutions).

The query times are expressed in seconds, and are usually averaged over several runs. The time needed to generate the compiler from the interpreter (i.e., performing the second Futamura projection [18]) was $0.040s$. The prior binding-time analysis was performed (once and for all) by hand using LOGEN's new graphical interface that allows easy annotation and provides colouring feedback on static and dynamic parts.An automatic binding-time analysis is in the final stages of implementation (as can be guessed from the screenshot in Section 5.4) and it will hopefully be able to annotate our interpreter automatically. However, it is acceptable to perform the BTA by hand, as the annotation only has to be generated once and it is independent of the database as well as the access control policy. To achieve the good results it was essential to follow the approach from [36]. Timings for Ciao Prolog were obtained using version 1.11 (patch 164) on a Powermac G5 Dual 2.5 Ghz, 4.5GB of RAM. Timings were obtained on a Powerbook G4 1Ghz, 1GB SDRAM, with SICStus Prolog 3.11.0 and Mac OS X 10.3.2 (this machine is slower than the one for Ciao; for example the query Q_4 runs in 3.4 s rather than 9.64 s on the Powermac G5 Dual). The reason for the initial program being slower in Ciao (Ciao is generally comparable to SICStus) is that the source is missing a meta-predicate declaration that Ciao (which implements higher-order in a different way from SICStus) requires. However, we have decided to use an identical initial program because it allows to observe better the impact of the specialization, which removes higher-orderness from the program. Runtimes for XSB were obtained on the same machine using XSB Prolog 2.6. In our experiments, we make use of XSB's distinctive feature: it terminates for both recursive and non recursive datalog programs. This mechanism is known as *tabling* in XSB Prolog [49], and has been proved very useful in deductive databases. Tabling allows, for instance, the evaluation of query Q_3, which only XSB Prolog can run ensuring termination.

Table 5 shows how much overhead is introduced by the access control policy. For example, query Q_3 that takes 1.08 seconds to retrieve information takes an extra 0.38 seconds when access control is performed. Ideally, we want to minimise the overhead introduced by the $RBAC^P_{H2A}$ policy. By specialisation of the meta-interpreter, we achieve a speedup that considerably reduces this overhead, as illustrated in Table 6. It can be observed that after applying the LOGEN tool, the average retrieval time is improved by a factor of up to 42. In all cases the retrieval time after specialisation falls between the average times of the two previous approaches, i.e., *with* and *without* access control.

There is, of course, a penalty introduced by this approach: the *specialisation time*, i.e., the time it takes *logen* to figure out a specialised version of the meta-interpreter with an $RBAC^P_{H2A}$

Query		*With $RBAC^P_{H2A}$*	*Without $RBAC^P_{H2A}$*	*Overhead*
Q_1	*(Ciao)*	*1.530 s*	*0.003 s*	*1.527 s*
	(SICStus)	*0.135 s*	*0.003 s*	*0.132 s*
	(XSB)	*0.100 s*	*0.000 s*	*0.100 s*
Q_2	*(Ciao)*	*4.490 s*	*0.013 s*	*4.477 s*
	(SICStus)	*1.372 s*	*0.004 s*	*1.368 s*
	(XSB)	*0.100 s*	*0.000 s*	*0.100 s*
Q_3	*(XSB)*	*1.460 s*	*1.080 s*	*0.380 s*
Q_4	*(Ciao)*	*23.08 s*	*0.060 s*	*23.020 s*
	(SICStus)	*9.640 s*	*0.060 s*	*9.580 s*
	(XSB)	*0.109 s*	*0.010 s*	*0.099 s*

Table 5: Average retrieval times for the non-specialised case.

		Non-aggressive specialisation			*Aggressive specialisation*		
		spec.	*average*	*speedup*	*spec.*	*average*	*speedup*
Query		*time*	*runtime*		*time*	*runtime*	
Q_1	*(Ciao)*	*0.010 s*	*0.040 s*	*38.3*	*0.010 s*	*0.003 s*	*510*
	(Sicstus)	*0.010 s*	*0.007 s*	*19.3*	*0.010 s*	*0.003 s*	*45*
Q_2	*(Ciao)*	*0.010 s*	*0.160 s*	*28.1*	*0.010 s*	*0.010 s*	*449*
	(Sicstus)	*0.010 s*	*0.032 s*	*42.9*	*0.010 s*	*0.004 s*	*343*
Q_4	*(Sicstus)*	*0.010 s*	*0.950 s*	*10.2*	*0.010 s*	*0.060 s*	*121*

Table 6: Retrieval times (running in Ciao and SICStus) for the specialised case.

policy. However, Table 6 shows that adding together the *average runtime* and the *specialisation time* does not exceed the original times. By adjusting the annotations (i.e., marking more calls as unfoldable), a more aggressive specialisation can be obtained. This is shown in the right-hand columns of the table, where Q'_i is specialised; the same query as Q_i, but taking the $senior\_to$ clause into account as well (which is likely to remain unchanged for a long time).

Table 7 shows how the previous results compares when the Prolog Engine includes tabling. It can be observed that for the more aggressive criteria the time is reduced and even reaches, in most cases, the ideal without-access-control figure aimed (i.e., overhead=0). For query Q_3 the specialised interpreter (see Figure 15) is actually almost identical to the database without access control. We believe the fact that our specialised interpreter runs slightly slower is probably due to

	Non-aggressive specialisation			*Aggressive specialisation*		
	spec.	*average*	*speedup*	*spec.*	*average*	*speedup*
Query	*time*	*runtime*		*time*	*runtime*	
Q_1	*0.010 s*	*0.026 s*	*3.85*	*0.010 s*	*0.010 s*	*10*
Q_2	*0.010 s*	*0.024 s*	*4.17*	*0.010 s*	*0.008 s*	*12.5*
Q_3	*0.010 s*	*1.220 s*	*1.20*	*0.010 s*	*1.130 s*	*1.29*
Q_4	*0.010 s*	*0.016 s*	*6.81*	*0.010 s*	*0.013 s*	*8.38*

Table 7: Retrieval times (running in XSB) for the specialised case.

caching issues. We have thus actually achieved what is called “Jones optimality” [27, 28, 37, 36] (called the “optimality criterion” in [28]). The only drawback of the aggressive specialisation is that each time $senior\_to$ changes there is an overhead of $10ms$ for specialisation (as well as the time needed to load the new specialised interpreter, which was around $10ms$ in our experiments). Since this clause does not change very often, we are not paying a high price in terms of access control flexibility.

Observe that `holds_read_rule__2` is isomorphic to the `cycle` predicate, hence Jones-optimality [27, 28, 37, 36] has been achieved.

Our testing is based on the scenario where a user inherits all access privileges on all objects (i.e., logical consequences) from the most junior role in the $RBAC^{P}_{H2A}$ role hierarchy. In practice, access request evaluation will involve significantly less permission inheritance, and user access to data objects will be far more constrained than we have considered in our testing. We consider only this scenario because access control requirements will be highly application specific. The more specific access control restrictions that will apply in practice will enable LOGEN to specialise access control programs to a much greater extent than we have considered, and therefore more impressive speedup can be expected in practical applications.

5.5 Future Work

There are a number of additional issues to investigate in the context of optimising access requests on policy information in P2P and B2B applications. It would also be interesting to apply our approach to emerging access control models for controlling access to Web resources (see, for example, [5]). We intend to investigate these issues in future work.

```
bench__0 :-
    ensure_loaded(database_cycle),
    abolish_all_tables,
    cputime(A),
    b2__1,
    cputime(B), C is B-A,
    print(C), nl.
b2__1 :-
    seniorto(r1, r53),
    holds_read_rule__2(_, _),
    fail.
b2__1.
:- table holds_read_rule__2/2.
holds_read_rule__2(A, B) :-
    seniorto(r1, r53),
    p(A, B).
holds_read_rule__2(A, B) :-
    seniorto(r1, r53),
    holds_read_rule__2(A, C),
    seniorto(r1, r53),
    p(C, B).
```

(a) The non-aggressive approach: annotating the $senior\_to$ clause of `permitted/3` as a *rescall*, so that it is not evaluated in the specialisation process.

```
bench__0 :-
  ensure_loaded(database_cycle),
  abolish_all_tables,
  cputime(A),
  b2__1,
  cputime(B), C is B-A,
  print(C), nl.
b2__1 :-

  holds_read_rule__2(_, _),
  fail.
b2__1.
:- table holds_read_rule__2/2.
holds_read_rule__2(A, B) :-

  p(A, B).
holds_read_rule__2(A, B) :-

  holds_read_rule__2(A, C),

  p(C, B).
```

(b) The aggressive approach: considering the $senior\_to$ clause as *unfold*, i.e., being computed in the specialisation, the resulted program may be more efficient.

Figure 15: Specialised interpreter for Q_3. The actual code obtained from the specialiser, for both non-aggressive **(a)** and aggressive **(b)** specialisation.

6 Conclusions

In this deliverable we have presented our first cycle of experiments that explore the benefits of specialisation and program analysis within a pervasive environment. This first cycle contained four cases studies, two of which are complete and two of which are still in progress. The scope and nature of our results has varied from case-study to case-study. In this section, we tie together those results, make overall conclusions for the current generation of tools, and present the lessons learnt in selection of experiments, and how they will be applied to our next round of case-studies.

6.1 Case-studies

The Timing Analysis case-study (Section 3) has not produced any direct results in the application of specialisation tools, but it used specialisation techniques in order to build a stand-alone tool. It has advanced our knowledge in the problem domain, and resulted in a tool that is directly applicable to real-world problems in pervasive computing. This tool has already discovered errors in existing pervasive systems, and allows us to verify the correctness of the fix. This case-study gave us valuable initial experience in applying program transformations to a useful problem in pervasive computing.

The Access Control Verifier case-study turned out to be a textbook example of how specialisation (and the ASAP tools in particular) can be employed in order to speed up execution of a program. We show that we can reduce execution time to acceptable levels. The overhead can be reduced to close to zero. This case-study has been applied both to ASAP tools, and to external tools for comparison. Tabling, as implemented in XSB reduces the gains of specialisation, limiting speed-ups to single-digit figures.

The Timing Analysis and the Access Control Verifier case-studies have produced valuable experience and direct results. They were both case-studies at the lower range of risk; producing well understood results but not being too adventurous in how far they pushed the state of the art. Both studies have contributed as much to the progress of the project as they will and we will not be actively pursuing further results within them. The higher-risk case-studies were the PIC-Emulator and the Precision Interpreter. Both of these case-studies have produced encouraging initial results that show that we will progress the state of the art by continuing to develop them in the later stage of the project.

The Precision Interpreter case-study (Section 2) targets a prevalent problem in pervasive system design. We have shown promising initial results in this domain. Our analysis of the problem is developed in the native language of the tool-set, and has shown more flexibility and better results than the previous work in the field. Our use of an interpreter to model the problem is a novel approach within the domain. We expect there to be much more scope in that case-study, in particular on the code generation side; preliminary experiments with CiaoPP show that we should be able to specialise the Precision Interpreter.

The PIC Emulator case-study (Section 4) has so far suggested that it would be applicable to a wide range of static analyses. Currently we have successfully specialised the PIC interpreter with respect to real programs from the pervasive domain, including a program that detects steps using an accelerometer. The residual programs that we have produced appear to be amenable to liveness and deadness analysis, redundant code, unreachable code, and even to subsume the timing analysis (Section 3) within a more general framework. We have manually identified these analyses within the residual program, but we have yet to automate the analysis.

6.2 Results: Pervasive Requirements

There are three pervasive requirements that we set out in the introduction. Below we identify how the case-studies have contributed to those goals.

Memory footprint The access control verifier has not yielded any improvements in memory footprint that we measured.

The PIC emulator has shown that unused memory within a target program can be detected. The first "real" PIC program to be analysed was a 321 instruction program which is a step detector that uses an accelerometer. Specialisation followed by analysis showed that 56 instructions (17 percent of the code) were unreachable. Memory savings will mean that the code can run on a smaller device. The memory savings will come both from unreachable parts of the program as just mentioned, and unused variables/registers, and using liveness analysis to use registers more efficiently. Preliminary results on liveness analysis for the PIC are promising.

Additionally, the Precision Interpreter can reduce the memory requirements of its target programs. Even though in absolute terms the savings are modest (bytes per variable), since our target architecture has only 68 bytes of memory the savings are significant.

Correctness We have used the Timing Analyser to verify the behaviour of legacy code. The tool identified errors within that legacy code that were traced to malfunctions in the system. We were able to fix those errors and to verify the correctness of the fix.

Power consumption The Access Control Verifier case-study has managed to significantly reduce the time required for simple access control. Specialisation can speed up access control to a zero-overhead process, at a marginal specialisation cost.

The Precision Interpreter successfully analyses code and identifies the precision at which operations have to take place. We have not yet *quantified* how few instructions will be required, as this will result from the study of code generation that we will undertake in the second round of case-studies.

The PIC emulator could identify redundant instructions, but has yet to do so in the first test cases that we have tested it on.

6.3 Results: Tools and Methods

A common theme in the case studies is the use of Prolog to interpret and emulate classes of pervasive applications and machines, followed by specialisation to obtain Prolog programs corresponding to specific applications and programs. The tool-set has successfully specialised the interpreters that we have needed for the first stage of the project. In particular the offline partial evaluator LOGEN, with its Binding Time Analysis tool, proved able to handle sizable interpreters. We got a very good result in the Access Control Verifier case-study using LOGEN. Also, the tools have been successful in specialising the PIC Emulator and the Precision Interpreter. We found the development of the Precision Interpreter very easy, due to the use of Prolog and the specialisation tools. In particular, we were able to focus on defining a correct and clear formulation of the Precision Interpreter, rather than an efficient formulation, because we knew that the redundant code is going to be specialised out. We expect this to be the case on other programs.

We found another problem whilst the pervasive systems developers were learning to use the ASAP tool-set. It turns out that they tend to write Prolog code in a very different style to most logic programmers. This naive style of programming resulted in very declarative code that was highly inefficient to execute and specialise. The normal technique to increase the efficiency of the code would be to reduce the number of choice-points through the introduction of (green) cuts. It would be desirable for the tool-set to perform this task automatically; given a set of entry points to a module the tool-set could determinise the code, and produce the less declarative but more efficient implementation.

Finally, one of the lessons that we learnt whilst developing the Precision Interpreter was that in order to express the problem clearly we were manually staging the code. In order to increase the modularity of the code it would be desirable to place related functionality together. One method of achieving this (and improving the clarity of the code) would be to to introduce manual staging constructs into the language. If the BTA of the specialiser understood these syntactic constructs then it would allow the programmer to “massage” the BTA and effectively specialise more complex code.

The case-studies provided a genuine test of the scalability of the analysis and specialisation tools, since the interpreters and their specialisations are definitely not toy problems, but contain in some cases thousands of lines of Prolog code, with high-arity predicates. The results were generally encouraging though optimisation of analysis algorithms and data structures is a clear requirement. For numeric domains, we plan to introduce an external polyhedron library (the Parma Polyhedron Library) rather than CLP, while for symbolic domains, more efficient representations of pre-interpretations, and the use of BDD representations are being investigated. However in general the approach of using *general purpose* analysis and specialisation tools looks capable of achieving significant results.

The requirement for effective analysis tools over numerical domains such as intervals and convex hulls was underlined in more than one case-study, including the Precision Interpreter, Timing Analyser and PIC interpreter. This requirement will be followed up in the future work on the analysis tool-set.

6.4 Future work

We do not plan to directly extend the Timing Analyser within the scope of the ASAP project, as it has provided the experience and results that we feel are relevant and the effort to do so would be better invested in a different case-study. However, the tool itself is scheduled to be re-written in a language applicable to the tool-set and investigated with the ASAP tools, outside the project. Also, we will be intrigued to see whether the functionality of the timing analyser can be subsumed by the PIC Emulator.

We will conclude the research on the Precision Interpreter in the second round of case-studies. Our focus will be on the generation of low-level executable code for the PIC micro-controller. We believe that the strength of our approach will best be demonstrated by tackling such an ambitious compilation target. As this case-study is well suited for real-world problems in pervasive computing we believe that such a result would have a very positive effect on research in pervasive computing. We will measure this result by a comparison of the output code against both hand-written code and other compilers that do not translate from such a tightly defined domain-specific-language.

While our main focus will be on the specialisation of the Precision Interpreter in order to produce the most efficient program that we can, there are other interesting areas of research within the case-study that can be pursued if there is sufficient time. As we have stated in Section 2.5 there is a large scope for optimisation if we can deduce the static properties of the program running within the interpreter. This is a difficult analysis problem as it requires information from separate calls at different levels of interpretation to be unified. Such an analysis may also have an effect on the Stream Interpreter.

We intend to continue the work undertaken in the PIC Emulator in the second round of case-studies. One interesting area of research that we will undertake is the analysis of interactions between the PIC micro-controller and its external environment. We will experiment with the "backwards analysis" method [21] in this problem, and how it can be used to detect possible error conditions at run-time. We will also try to analyse numerical properties, including looking for constant values in registers, and relationships between the clock and frequency of input values. The initial work on register liveness is promising and we aim to develop this further, as well as the optimisations of register usage that become available following liveness analysis.

An interesting problem revealed itself with the Stream Interpreter case-study. Although from the point of view of a pervasive designer this was quite a simple case-study, the depth of analysis required in order to specialise an interpreter of this type was too ambitious for the tools at the time. The difficulty in analysing the Stream Interpreter is that it provides a novel view of redundancy in the code. Rather than an analysis that operates on a pure division between static and dynamic data, this problem requires an analysis capable of detecting partially static data over varying time frames. The straightforward method to attack this problem is to aggressively unfold the entire program in order to make explicit the semi-constancy across loop iterations. But using this approach requires us to retain enough information to refold the program into a form suitable for execution.

However, since we first tried the stream interpreter, the tools have improved and we have a better understanding of how to formulate the stream interpreter. We are currently investigating ways of implementing this semi-constancy that are not specific to this particular case-study but which will also have value when applied to other programs, and we plan to attempt the stream interpreter in the second cycle of case studies. Another case study that we wish to study in the second round is the specialisation of matrix operations used in the Kalman Filter.

References

[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] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 283–295. ACM Press, 1990.

[3] J. Armstrong, S. Virding, and M. Williams. Use of prolog for developing a new programming language. In *Proc. 1st Conf. on The Practical Application of Prolog*. Association for Logic Programming, 1992.

[4] S. Barker. Protecting deductive databases from unauthorized retrieval and update requests. *Journal of Data and Knowledge Engineering*, 23(3):231–285, 2002.

[5] S. Barker. Web usage control in rsclp. In *Proc. 18th IFIP WG Conf. on Database Security*, 2004.

[6] S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. on Information and System Security*, 6(4):501–546, 2003.

[7] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J.P. Gallagher, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *Springer-Verlag Lecture Notes in Computer Science*, pages 204–223, August 1996.

[8] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM TODS*, 23(3):231–285, 1998.

[9] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proc. IEEE 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, 2002.

[10] A. Briney. Information security 2000. *Information Security*, pages 40–68, 2000.

[11] K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[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] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[15] C. Date. *An Introduction to Database Systems*. Addison-Wesley, 2003.

[16] 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.

[17] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proc. of the 11th Annual Computer Security Applications Conf.*, pages 241–248, 1995.

[18] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[19] J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

[20] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, LNCS. Springer-Verlag, January 2002. Accepted for publication.

[21] J. P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe, editor, *Proceedings of the International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2003)*, LNCS. Springer-Verlag, 2003. (to appear).

[22] J. P. Gallagher and K.S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, Springer-Verlag Lecture Notes in Computer Science. Springer Verlag, 2004. (to appear).

[23] J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher Order and Symbolic Computation*, 14(2,3):143–172, 2001.

[24] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, CO, June 1998.

[25] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, May 2000.

[26] S. Jajodia, P. Samarati, M. Sapino, and V.S. Subrahmaninan. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.

[27] Neil D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming*, LNCS 443, pages 639–659. Springer-Verlag, 1990.

[28] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[29] 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.

[30] S. Kim, K. Kum, and W. Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Transactions on Circuits and Systems-II:Analog and Digital Signal Processing*, 45(11):1455–1464, November 1998.

[31] 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.

[32] 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.

[33] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Electr. Notes Theor. Comput. Sci.*, 30(2), 1999.

[34] M. Leuschel, J. Jorgensen, W. VanHoof, and M. Bruynooghy. Offline specialisation in prolog using a hand-written compiler generator. *TPLP*, 4(1):139–191, 2004.

[35] M. Leuschel and D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *JLP*, 36(1):149–193, 1998.

[36] Michael Leuschel, Stephen Craig, Maurice Bruynooghe, and Wim Vanhoof. Specializing interpreters using offline partial deduction. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.

[37] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.

[38] K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[39] Daniel Menard, Daniel Chillet, Fran&#231;ois Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 270–276. ACM Press, 2002.

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

[41] 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.

[42] 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.

[43] A Pnueli. *Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends*, pages 510–584. Springer-Verlag New York, Inc., 1986.

[44] T. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufmann, 1988.

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

[46] 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.

[47] 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.

[48] K Sagonas, T. Swift, D. Warren, J. Freire, and P. Rao. *The XSB System Version 2.0, Programmer's Manual*, 1999.

[49] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.

[50] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. 4th ACM Workshop on Role-Based Access Control*, pages 47–61, 2000.

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

[52] Walid Mohamed Taha. *Multistage programming: its theory and applications*. PhD thesis, 1999. Supervisor-Tim Sheard.

[53] 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.

[54] 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.

[55] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W Liao, C.-W. Tseng, M. Hall, M. Lam, , and J. Hennessy. SUIF: A parallelizing and optimizing research compiler. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.