IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D29 (WP5c): Refinement of the Machine Level Analysis

Due date of deliverable: 31st August 2008
Actual submission date: 31st August 2008

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: St Andrews University (previously: AbsInt GmbH) Revision: 1.30

Purpose: The purpose of this deliverable is to show how low-level and high-level analysis can benefit from each others' results and how they can potentially be integrated.

Results: The main results of this deliverable are that cost information can conveniently be regarded in the high-level analysis by parameterised cost formulae and that splitting the cost of complex bytecode instructions into several constants can effectively liberate the analysis from the need of having run-time information like sizes or loop bounds. Extending low-level analysis to larger parts of the code turned out not to be useful; a combination of low- and high-level analysis is required.

Conclusion: The main conclusions are that the amortised analysis and the translator into the core language can and should be further developed with the aim of tackling overloaded operations on dynamic data structures, with size inference as a potential further source for improvements.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⁎
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Refinement of the Machine Level Analysis

Christoph Herrmann <ch@cs.st-andrews.ac.uk>,
Steffen Jost <jost@cs.st-andrews.ac.uk>
School of Computing Science, Univ of St Andrews, St Andrews, Scotland

Abstract

This deliverable is about the combination of high-level amortised analysis as reported in Deliverable D14 [7] about WCET analysis and the low-level machine analysis whose possibilities are described in Deliverable D06 [3], dealing with the extensions of the interface language AIS.

Major Revisions		
Revision	Date	Changes
1.29	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.27	31 Aug. 2008	initial version

Contents

1 Introduction

The current deliverable D29 inhabits WP5c which depends upon WP5b/D06 [6] ("Extended AIS interface language") and updates the WCET results from WP3a/D14 [7] ("Report on WCET analysis"). Its goal is to combine the information gained in the high-level and low-level analyses to achieve better overall analysis results for entire Hume functions or boxes.

Information which is required for the analysis, comes from knowledge about the target machine architecture, gained by the low-level analysis, and from knowledge about the program semantics, gained by the high-level analysis. Both ways of analysis are very different and combining them into a single analysis would not only worsen the comprehensiveness but also lead to an explosion of computation time to carry out the analysis. The reason for the latter is that the same, or very similar tasks at the low level would have to be repeated thousands of times if integrated as a software component in the high-level analysis of a Hume program of medium size.

Instead, we break the analysis computationally into two parts where the logical interface is roughly at the level of the HAM bytecode instructions. Most HAM bytecode instructions have been analysed at the machine-code level (low-level) using the AbsInt GmbH's aiT tool [1], with the result that a particular execution time for them can now be given, either as a constant or as a linear formula expressed in parameters. This cost formula can be added by the high-level analysis to the cost constraint system each time the according HAM code needs to be executed, without the need to repeat the low level-analysis. The high-level analysis itself can deliver a solution which is parameterised in the number of occurrence counts of constructors in the input, i.e., based on semantic knowledge about the program as in the drilling robot example [4]. The worst-case execution times for HAM instructions are listed in Appendix A, together with artificial cost items which are used to shift part of the costs of a HAM instruction to an earlier HAM instruction which incurred part of the computation load for the later instruction. This is both to simplify assignment of costs and maintaining a system of linear constraints.

The following sections deal with the different approaches we applied. In Section 2 we exploit context information to improve the low-level analysis. Section 3 describes how low-level information is factorised according to loops in the control flow graph depending on program parameters and exposed to the high-level analysis. Section 4 discusses the impact of type information. Tail-call optimisation is described in Section 5. The study in Section 6 evaluates the extension of the low-level analysis on larger parts of the program. The report ends with a summary of the results.

2 Exploitation of context information to guide low-level machine analysis

Although the worst-case execution time analysis tool aiT can sometimes deliver surprisingly good results completely automatically, it is also often the case that it is not able to find a solution, especially when a loop bound is unknown, or that an analysis result is pessimistic. This can have the following reasons:

1. **Lack of context information.** Contents of registers are often influenced by the context in which a particular code sequence is used. This context information is not known by the aiT tool and thus, if it is not specified explicitly, cannot be exploited.

2. **Tradeoffs to make the analysis practically tractable.** Even if the complete description of the processor is available, carrying out a global analysis on the assembly code would not finish within an acceptable amount of time. This would basically result in specifying all potential combinations of register and memory values when the control reaches a certain point. Section 2.1 demonstrates how a manual analysis of the control flow graph can infer a loop bound that currently cannot be found by the automatic analysis, and how this knowledge can be made available to the aiT tool.

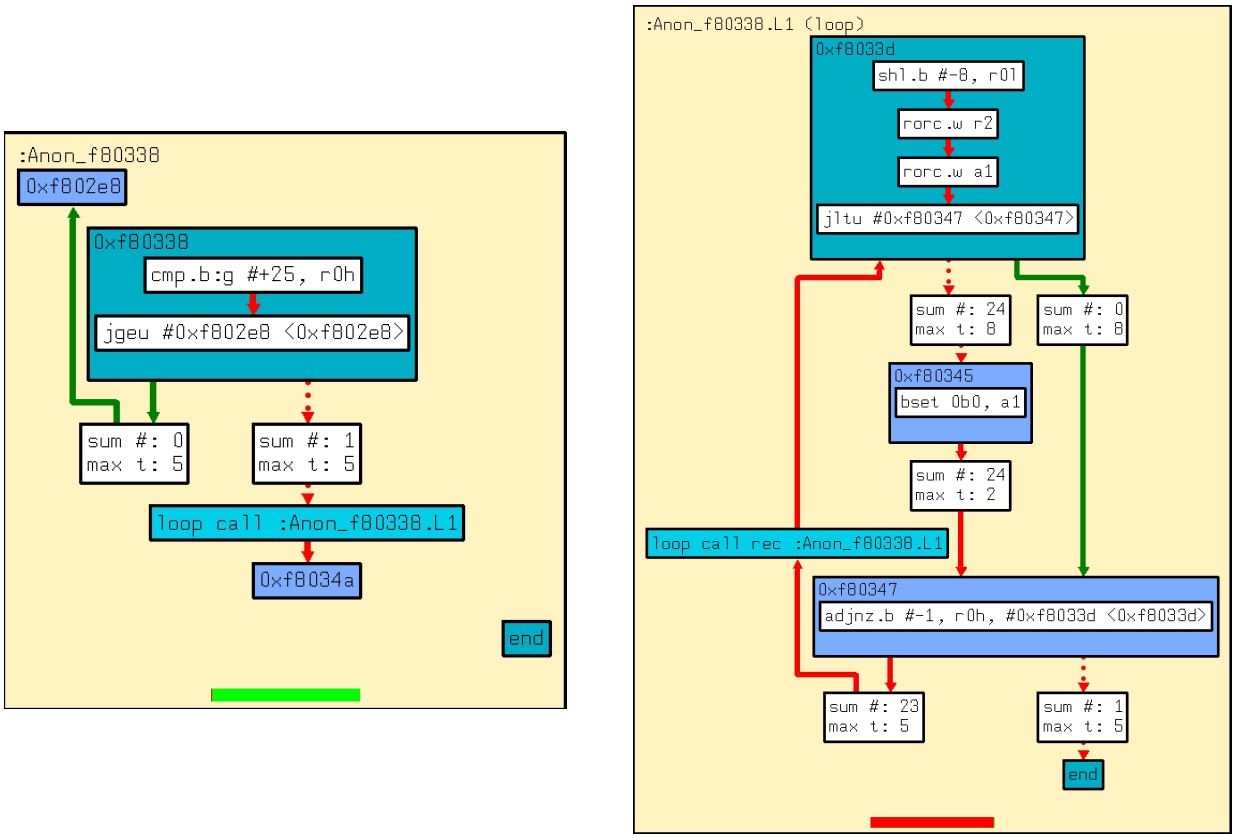

Figure 1: Shifting loop in floating point addition: (a) call, (b) body

Therefore, it is sensible for an analysis tool like aiT to accept external advice. The aiT tool delivers its results based on many parameters of the machine and the program under analysis. The values of these parameters can be specified in the aiT specification language (AIS). Some parameters, e.g., the kind of memory used are absolutely essential to deliver correct results. We do not discuss these parameters here further because their values are not free of choice. Other parameters specify strategies to balance analysis time with precision, e.g. the degree of unrolling loops, or provide information about the program under analysis, e.g. bounds on the number of loop iterations.

An example of the former kind is the specification of a context; an example of the latter the determination of loop bounds in the floating point library.

2.1 Exploiting loop bound information in the low-level analysis

Figure 1 shows the control flow graph of the mantissa shifting loop in the library routine for floating point addition, at the left side (a) the context of the loop and the right side (b) the loop body. The aiT tool expresses loops as tail recursive calls for analysis reasons, reentry in the loop body is thus named `loop call rec:`. We obtain a similar picture when all basic blocks start at even addresses, but the execution times for two of the blocks are one clock cycle less. Thus, we conclude that the WCET per iteration of the loop body is 15 (=8+2+5) machine cycles.

From part (a) of the figure we obtain that the loop body can be entered with a value of the register `r0h` which is at most 24. Thus, we can declare in the AIS specification language that this loop as at

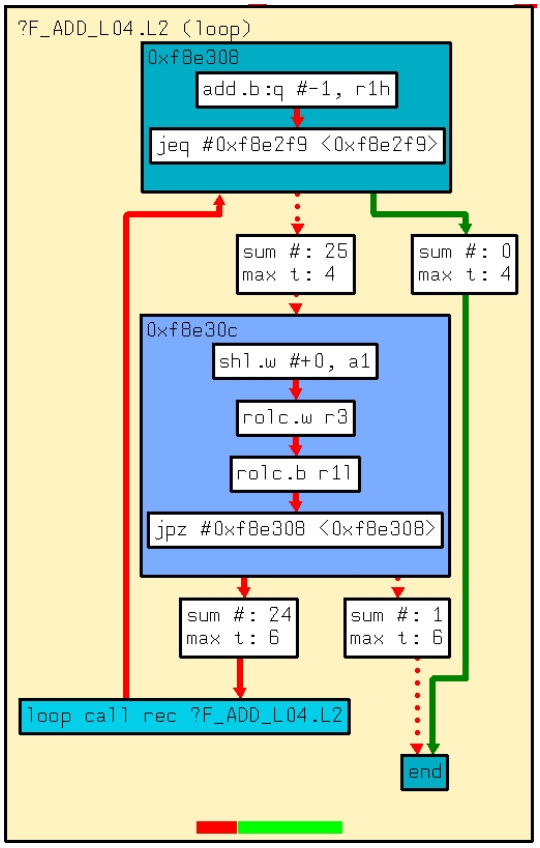

Figure 2: Loop `F_ADD_L04.2` in floating point library

operation	+	−	*	/	sqrt	sin	cos
machine cycles	934	938	356	962	5177	11744	11595

Table 1: Times for floating point operations in Hume

most 24 iterations, with the decrement of `r0h` and the test for exit performed at the end. The `sum#` numbers in part (b) of the figure are already a result of this annotation.

There is also another loop `F_ADD_L04.2` shown in Figure 2 which can only have 25 iterations due to the sequence of rotations. We add an appropriate specification as well.

For another loop required by floating point division the aiT tool itself can derive the loop bound. This is because this value had been written into a register used for decrement explicitly, while in the examples mentioned above the limits resulted from a combination of (1) a failing test for not being ≥ 25 with the additional information that the value is not negative and (2) a sequence of rotations via two registers of 16 and 8 bits with the test for the zero flag. All other loops in the floating point library have been checked and found that 33 is an upper bound for their iteration count. Table 1 shows the results after annotating the shifting loop with 24 iterations, `F_ADD_L04.2` with 25 iterations and the other loops with 33 iterations.

Note that the WCET value of 15 for an iteration of the shifting loop must not be subtracted in order to derive values in case that the number of iterations is less. In fact, in the context of a Hume floating

point addition the path going through `bset` is infeasible and the right green path chosen, which leads to a value of 13 cycles per iteration. This is automatically regarded in the Table 1, since the values there have been obtained by a separate analysis for the machine code for each HAM code separately.

The example program we used to analyse the Hume operations cause the blocks of the loop `F_ADD_04.2` to be placed at odd addresses which is correct for WCET of that loop, but with this program the blocks of the shifting loop are placed at even addresses which leads to 2 machine cycles less per iteration. To compensate this, the values in Table 1 above had been corrected by adding machine cycles to the values the aiT tool delivered, in particular 48 (=2*24) cycles for addition and subtraction, 144 (=3*48) for `sqrt` and 432 (=9*48) for `sin` and `cos`.

2.2 Gaining a proof of linearly bounded sizes by the art3 tool

Loop bounds are especially influenced by sizes of data structures.

The amortised analysis tool **art3** developed at LMU and UStA does not directly compute information about the sizes of data structures like lists, which could then be used to obtain loop bounds, e.g., for traversing these structures. However, the amortised analysis applied to heap consumption, when it succeeds, has proven that the sizes of function outputs are bounded by a linear expression in the sizes of the function's input data structures. However, we do not obtain such a proof for all linear size-transforming functions, because the method is based on heap consumption and the heap consumption can be more than linear even if the result is linear.

We can obtain a proof of a linear size-transforming function for the following list functions:

- duplicating each element of a list
- appending two lists
- map: applying a function to each element
- scanl: computing a generalised prefix sum
- concatenation of a list of lists
- separating odd/even elements of a list
- inserting an element into an ordered list
- merging two lists alternatingly

When potential is assigned to numerical values (art3 option `--ap`), it is also possible to include values of numbers in the linearity analysis:

- length of a list, where the analysis result is a linear expression for the number returned itself
- gen: generating a list of a given length
- the composition (gen $\circ$ length)

However, the following functions have a linearly bounded output size although their total heap consumption is not linear:

- list reversal without use of an accumulator
- sorting by successive insertion
- merging two sorted lists

In Hume, heap space is not reclaimed during execution of a box. Therefore, this method is not applicable to prove that the size relation between inputs and outputs of functions is linear.

2.3 Computing loop bound information by profiling linearly bounded functions

In the previous subsection we have seen that as a byproduct of the amortised analysis we can gain the information that the size of a function's output data structure is linearly bounded in the sizes of it's input. In that case, a simple interpolation of the function with test input data can obtain the precise mapping between the sizes of input and output data structures. The input/output size relation is of importance for determining loop bounds and somehow complementary to the amortised analysis, which abstracts from this information by expressing costs in terms of occurrences of constructors.

3 Abstracting from structural parameters in the low-level analysis

3.1 Traversing lexical scopes

A local definition in a Hume program, e.g. by a let expression, introduces a new scope which is taken account of by a new frame on the stack. If data from a previous stack frame is accessed, first a sequence of stack frame links has to be traversed to reach the frame in which the data is actually located.

E.g., the HAM instruction `PushVarF n d` accesses the stack object located in the nth enclosing lexical scope (in the nth previous frame) at offset d.

The example shows the C code for `PushVarF 2 1`:

```
SP lfp = slp;
lfp = stack[PREV_LINKF(lfp)].sp;
lfp = stack[PREV_LINKF(lfp)].sp;
stack[inc_sp()] = stack[VARF(lfp, 1)];
```

The aiT analysis obtains that following a stack frame link takes 11 cycles while all the rest of the code takes 35 cycles. The cost thus depends on the number of frames traversed but not on the offset within the frame. Consequently, only the first parameter is contained in the cost expression:

```
Tpushvar n | n>0 -> 11*n+35
```

The cost expression is available to the high-level analysis tool where the required value of n is known and substituted.

3.2 Treatment of Closures

For some kinds of analysis, both high-level and low-level information needs to be available. An example is the treatment of closures.

As typical for functional languages, Hume permits functions to be applied in instalments. Until a function has received all its arguments, it cannot commence the evaluation of its body. In that case a closure is created in which the values of all arguments so far provided are stored. The HAM code provides the following instructions to deal with that:

- `MkFun` *name args provided*:
  creates a closure for a function with *args* arguments and fills it with as many as stated in *provided*.
- `Ap` is applied when a closure is returned and there is already an additional argument on the stack (this is called overapplication). It is then checked by the library function `callVar` whether adding this argument would make the closure fully applied. If so, the function is called, otherwise a pointer to a new closure including the additional value is left on the stack. The cost constant

for `Ap` covers only the overhead for checking the additional argument. The cost for potentially selecting and calling the function is by adding the cost for a CallVar in the analysis at some other point.

Another functional language feature present in Hume is to pass a function as an argument to another function. The caller creates a closure of this function using the `MkFun` instruction and the callee uses the HAM instruction `CallVar` with the pointer to the function stored in the formal parameter.

Both `CallVar` and `Ap` make use of the library function `callVar` which depends on three run-time values. Two of them are passed as parameters: the number of arguments provided and a pointer (index) to the function to be called when all arguments are provided. The third run-time value dependency is the number of chunks in which the closure arguments are splitted due to the number of under-applications in which it has been created.

To avoid quadratic cost formulae and dependences of run-time values in the analysis –either of them would lead to a failure of the available methods– each chunk of closure to be filled with arguments is given a separate cost constant. Fortunately, the high-level analysis knows the number of closure arguments provided.

As a consequence, we end up with three parameterised cost formulae for partial application and application of functional variables:

- `Tmkfun provided -> 166+60*provided` is applied whenever a functional closure is created, when passing a function as a parameter or in the case of an underapplication.
- `Tunwindchain provided -> 49+51*provided` is applied when values stored in a closure are copied onto the stack in favor of a function to be applied soon. The parameter `provided` is the number of values that have to be copied. This number is not known at the point of copying but it was known when the closure was created. Therefore the analysis annotates the type with the value of the parameter `provided` and the cost is assigned each time this closure unpacking is executed.
- `Tunwindfun provided funs -> 301+51*provided+15*funs` similar to `Tunwindchain`, but the function is now fully applied which triggers its execution. Before the evaluation of the functions body can happen, the appropriate code for the function has to be selected according to the function's identifier stored in the function variable. The value `funs` is the number of functions that are potential candidates for the selection.

Here, both high- and low-level information are required. At analysis, i.e., before the actual execution, only the level of operational semantics can provide information about how many arguments are provided and required for functional closure. On the other hand, machine level analysis is required to determine how many cycles each unwinding of a closure chain consumes.

4 Exploitation of type information

We distinguish exploitation of type information which is due to code specialisation, discussed in Section 4.1, and such which is done by analysis of polymorphically generated code, discussed in Section 4.2.

4.1 Generating specialised code for different types

User-defined functions can be overloaded, e.g., such that they work on integers as well as floating point numbers. But while an integer addition takes only 116 cycles, a floating point addition may require 934. The front-end compiler phamc implements a type specialisation and flags the HAM instructions appropriately. Depending on these type annotations, the backend ham2c of the Hume compiler generates specialised code for different types:

- integer addition:

```
sp--;
stack[sp-1].hp = mkInt((stack[sp].hp+1)->lv + (stack[sp-1].hp+1)->lv);
```

- float addition:

```
sp--;
stack[sp-1].hp = mkFloat((stack[sp].hp+1)->fv + (stack[sp-1].hp+1)->fv);
```

The C compiler knows the type of the addition operator from the type of its arguments: for integers, the `->lv` variant of a heap object is selected, which has type `long`, and for floating point values the `->fv` variant of type `float`.

For precision of the analysis, the type information is used to account for the right cost for the particular type.

```
Tcallbprim op IntTyp IntTyp     ->
  case op of  PlusOp->116; MinusOp->116; TimesOp->124; DivOp->206; ModOp->1294; ...
Tcallbprim op FloatTyp FloatTyp ->
  case op of  PlusOp->879; MinusOp->881; TimesOp->359; DivOp->959; ...
```

Generating type-specialised code has two advantages. Firstly it improves the performance of the code and as a consequence the chance to obtain a better cost bound. Secondly, it makes the analysis simpler and therefore additionally reduces the overestimation.

4.2 Improved analysis for generic implementations

Type specialisation can eliminate run-time decisions on values which are already known at compile time. Due to the amount of code duplication it is only useful upto a certain degree, e.g., for simple types like ints or booleans. To implement a comparison for compound data objects a type specialisation could lead to a significant increase in code size, e.g. if the function that compares vectors comes in different variants for different element types.

However, knowledge about the type can also be used in the analysis directly, by cutting branches in the data flow graph. Using the aiT tool, this can be achieved by telling aiT that the program points in the false branches are never executed. This method is effective when recursive calls can be excluded from the analysis. We have applied this method to the overloaded equality function for the case that the types are booleans and obtained a special cost of 320 machine cycles of the M32C processor.

We are thinking about using phantom functions in the Schopenhauer code that mimic the traversal of the recursive C implementation of the polymorphic comparison, using specific cost constants for the branches that have to be selected at each level due to the type. This would not require size information except from vectors, because the amortised analysis expresses cost in terms of occurrences of constructors.

5 Regarding tail-call optimisation in the analysis

In functional programming, repeated iteration is expressed by recursion. Usually, each recursive call would require additional stack space for local variables which remains allocated until termination of the call. When recursion in Hume is used to implement repetition with many iterations, like in the driller example, this is neither acceptable from the point of memory consumption nor necessary:

```
step (xpos,ypos,rest,aps) =
 case rest of
    []     -> (aps, pos_ok (xpos, ypos))
  | (A:xs) -> step (xpos,   ypos, xs, ((xpos,ypos):aps))
  | (L:xs) -> step (xpos-1, ypos, xs, aps)
  | (R:xs) -> step (xpos+1, ypos, xs, aps)
  ...
```

In an imperative language, one would express this control pattern by a loop visiting the elements of the list in turn. The counterpart to `for` loops are tail recursions on the functional side.

Tail recursion has the property that after returning from the recursive call no action needs to be taken, and the result is the result of the recursive call. This means that the local variables of the tail recursive call are not required any more after return, and during the called function instance they are not required anyway. Consequently, the stack frame for the local variables could be reused for the recursive call. The so-called tail-call optimisation achieves that by replacing recursive calls and the following copying and return instructions by a jump (goto) to the beginning of the function, as the following piece of code shows.

```
slp = stack[PREV_LINK()].sp;
fp =  stack[PREV_FP()].sp;
stack[ARG(0)] = stack[sp-1];
sp = fp+4;
rp = sp;
goto f_step_0;
```

The correction of the frame pointer results from pattern matching and let expressions which themselves require a new frame.

Two kinds of high-level analysis have to come into play here.

The first one integrated in the compiler front end `phamc` has to detect whether the recursion is actually a tail recursion. In that case, an appropriate HAM instruction is generated, aside from the non-recursive branches which terminate the function execution.

The second, resource analysis tool `art3` then has to exploit the fact that a call is tail-recursive by applying special rules for tail calls. In particular, this is achieved for the stack analysis by subtracting the space for the local variables at the time of the call and also assigning zero stack consumption for the return value.

6 Extending the range of the low-level analysis

Currently, our default mechanism for analysing Hume functions is based on the level of HAM instructions. We obtain timing costs for each kind of HAM instruction by analysing the machine code it is compiled to. These costs are then combined by the high-level analysis.

The machine-code analysis can exploit context information that evolves within the code sequence for a single HAM instruction, e.g. initialisation of a register with a particular value, but not information about the context of the HAM instruction. The aim of the experiments presented in this section is to estimate how high the gain would be if the low-level analysis would be extended to larger code parts, covering several or many HAM instructions. We choose right-hand sides with a few numerical operations and conditionals.

6.1 Example: model of a simple robot

As our case study we choose a model of a simple robot which moves on the plane. The robot can be given two commands at each step:

1. `Go` r: go forward `r` units of length
2. `Turn` a: turn orientation by an angle a in the range $\{-\pi, \pi\}$

The robot is modelled as a Hume function which is given an action and an old state and returns the new state of the robot. We are looking at two versions of the robot model which differ in the representation of the state and whether expensive trigonometric computations are assigned to the `Go` action or the `Turn` action. The first version `f0` applies these, i.e. sine and cosine functions in the `Go` action when the robot moves forward. Even if the orientation does not change, these values have to be recomputed.

In function `f0`, the state is represented as a triple `(x,y,phi)` in which `x` and `y` are the distances from the origin in two dimensions and `phi` is the orientation of the robot. E.g., if `phi`=0, then a step forward would add one unit to the `x` distance, and if `phi`=$\pi/2$ then one unit to the `y` distance.

```
f0 :: action -> (Distance,Distance,Angle)
             -> (Distance,Distance,Angle);
f0 action (x,y,phi) =
 case action of
    (Go   r) -> (x + r * cos phi, y + r * sin phi, phi)
  | (Turn a) -> let n = phi+a;
                    pi2 = 2.0 * pi;
                    n1 = if n < 0.0 - pi
                            then n + pi2
                            else if n > pi
                                    then n - pi2
                                    else n
                in (x,y,n1);
```

The second version `f1` applies sine and cosine functions when the orientation of the robot changes and stores these values in the state. This way, we can directly see the impact of sign and cosine functions on the cost of the individual branches and also test which of these two versions is more convenient for different sequences of actions.

fct	rule order	`Go` branch			`Turn` branch		
		analysed	measured	ratio	analysed	measured	ratio
`f0`	`Go,Turn`	28045	7666	3.66	6876	3532	1.95
	`Turn,Go`	28108	7720	3.64	6813	3505	1.94
`f1`	`Go,Turn`	5062	2916	1.74	30557	6840	4.47
	`Turn,Go`	5125	2998	1.71	30494	6829	4.47

Table 2: Analysis and measurements for variations of the robot program

```
f1 :: action -> (Distance,Distance,Angle,Factor,Factor)
             -> (Distance,Distance,Angle,Factor,Factor);
f1 action (x,y,phi,s,c) =
 case action of
    (Go   r) -> (x + r*c, y + r*s, phi, s, c)
  | (Turn a) -> let ... (as in f0) ..
                in (x,y,n1,sin n1, cos n1);
```

Another variation which we will apply is to swap the two branches in both programs. This will also have consequences on the total times for different sequences of actions.

6.2 Analysis and measurements for the robot programs

Table 2 shows analysis results using the art3 tool without any modifications and compare it with values measured on the Renesas board. This is done for both program variations `f0` and `f1` and the rule order `Go`,`Turn` or reverse. For each of these variants we show for each branch the results of analysis and measurement of the entire function, starting before the call instruction and ending after the sliding of the stack after the return instruction.

Although the analysis results already regard the special loop annotations from Section 2.1 there are still significant differences to the measurements. Table 2 reveals a significant difference between the branches in which expensive floating point operations are accumulated (sine and cosine both use 9 floating point additions) and the others. In fact we have constructed the case study due to a guess that this might be a main cause. While function `f0` has these expensive operations in the `Go` branch, `f1` calculates them in the `Turn` branch. We can see a factor of 3.6-4.5 between analysis and measurements when floating point operations are involved excessively but only 1.7-2.0 where they are not.

There are several reasons for the analysis/measurement gap for code with excessive floating point operations. The most important is that the measurements had been carried out with values in a normal range, especially those which fit the specification of the program; such values lead to loop bounds which can be a factor of 5-10 lower than the worst case which the code of the floating point library would permit. Another reason is that we have not specified all possible combinations of contexts for all loops occurring in the library code, and it seems that there is in principle no significantly better way than an exhaustive profiling, which is computationally infeasible.

Regarding that in Table 2 the best analysis result still shows a factor of 1.73 with respect to the measurements, we might wonder about potential major lacks of precision introduced in the high-level analysis. Section 6.3 demonstrates that there is at least no major lack of information which has not already been inherited from the low-level analysis.

fct	`Go` branch	`Turn` branch
`f0`	25740	4832
`f1`	3379	28035

Table 3: Applying aiT analysis to entire right-hand sides

6.3 Using values of low-level analysis of right-hand branch

The aim of this section is to explore whether an extension of the low-level analysis to larger parts of the code would lead to significantly better analysis results.

The expectations are fostered by the increasing amount of context information which could potentially be exploited by the aiT tool. We are not expecting much from effects of the architecture because the Renesas M32C processor does not have a cache and the number of its instruction pipeline stages, which is 4, is far below the number of machine instructions for many HAM bytecode instructions. However, we might obtain some benefit from loop and value analysis.

The code range to which we can extend the work of the aiT tool has practical limitations. Especially, the treatment of a large choice of program branches leads either to an excessive amount of time required for the analysis or need for low-level annotations. There is a significant gap for analysis time between the right hand side of a rule and a complete function. Given that the actual execution time is dominated by the right-hand side chosen it appears reasonable to delegate just the analysis of that part to the aiT tool.

We have analysed the worst-case execution time of the right-hand sides of Hume rules in functions `f0` and `f1`. The results are shown in Table 3. These times have been obtained with the same loop annotations for the floating point library as used to obtain the cost table used by art3.

Then, we have replaced the right-hand sides in the Hume programs by a void expression and assigned the time obtained by aiT explicitly to it, using the `<>`α`;"`β`"<>` annotation. Here, α is the control constant 9512 explained in the art3 manual which applies the appropriate accounting for the cost β into the constraint system. The values for β result from the analysis results gained by aiT shown in Table 3 minus the cost for constructing the empty tuple, which we needed to introduce here for syntactical reasons.

Function `f1b` is the version of `f1` when `Turn` appears first in the pattern matching. We have inserted the values for `f1` from Table 3 and subtracted 112 cycles for constructing the empty tuple, i.e. 28035-112=27923 and 3379-112=3267.

```
f1b :: action -> (Distance,Distance,Angle,Factor,Factor) -> ();
f1b action (x,y,phi,s,c) =
 case action of
    (Turn a) -> <>9512;"27923"<>()
  | (Go   r) -> <>9512;"3267"<>();
```

In Table 4 we compare the worst-case execution times of the original analysis based completely on the cost table for HAM instructions with the times we obtained when the analysis of the right-hand sides are delegated to the aiT tool.

To our surprise the analysis for the `Go` branch based completely on the cost table for the HAM instructions could only marginally be improved by delegation to the aiT tool. For the `Turn` branch we can at least achieve improvements upto 7.5% in the version without sine/cosine (`f0`) which is potentially be caused by exploiting the branch structure of the conditionals.

fct	rule order	`Go` branch				`Turn` branch			
		complete	aiT-RHS	diff	in %	complete	aiT-RHS	diff	in %
`f0`	`Go,Turn`	28045	27212	833	−2.97	6876	6367	509	−7.40
	`Turn,Go`	28108	27275	833	−2.96	6813	6304	509	−7.47
`f1`	`Go,Turn`	5062	5011	51	−1.01	30557	29730	827	−2.71
	`Turn,Go`	5125	5074	51	−1.00	30494	29667	827	−2.71

Table 4: Introducing aiT results for right-hand sides into art3

7 Summary

Delegating the analysis of right-hand sides to an extended low-level analysis has not obtained a useful gain, but it has increased confidence in the quality of the high-level analysis. Nevertheless, information about of special program branches is of particular use for an analysis of compositions of Hume boxes regarding particular cases, as it is exploited by other groups [2, 5].

Thus, it seems to be more successful to introduce context information in the low-level analysis by parameters of cost formulae then by extending the range of the analysis to larger parts of the program. For the same reason, an interface which combines high- and low-level analysis as software components does not seem to be useful. Manual inspection of control flow graphs can reveal much semantic information which can then directly exposed to the high-level analysis, including the analysis results for the different parts. Although low-level analysis tools like aiT might be sophisticated, their value analysis cannot cope with an exploding amount of case distinctions and contexts; and since they do not know the semantics of the source program, they cannot focus on the analysis of those values which are actually relevant to obtain loop bound information.

Thus, both low-level and high-level analysis are required. The low-level analysis to regard the properties of the machine and the high-level analysis to exploit the semantic information. The connection between both can be done via parameterised cost formulae, i.e., functions. The structure of these functions is given by the control flow graph obtained in the low-level analysis and these functions are then used in the high-level analysis, i.e., according to the operational semantics which is abstracted in a type system with effects, the appropriate cost function is selected by the high-level analysis, applied to the static parameters also obtained by the high-level analysis and the result forms part of the overall cost.

In the presence of higher-order polymorphic functions it is not even possible to do the analysis at the bytecode level. It is necessary to split the cost of bytecode instructions into different parts which are accounted for at different parts of the high-level analysis, namely there where the information for the values of the static parameters is available.

We have also seen the benefit of combining high-level analysis with the front-end of the compiler, since it provides information about types and properties like tail-recursion which can be used to generate more efficient target code and obtain tighter cost bounds.

References

[1] aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/ait`. 1

[2] Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering worst-case execution times for software components. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. 7

[3] AbsInt Angewandte Informatik GmbH. Extended AIS interface language. EmBounded Project Deliverable, March 2006. Deliverable D6. (document)

[4] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. 1

[5] Meng-Luo Ji, Ji Wang, Shuhao Li, and Zhi-Chang Qi. Automated WCET analysis based on program modes. In *AST'06*, pages 36–42. ACM Press, May 2006. 7

[6] S. Jost and K. Hammond. Validation of the Prototype Worst-Case Executation Time (WCET) analysis. EmBounded Project Deliverable, October 2007. Deliverable D16. 1

[7] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14. (document), 1

A WCET execution times

cost item	cycles	comment
Tap	34	additional overhead overapplication
Tcall	70	branching the call
Tcallbprim	→	see binary operators tables
Tcalluprim	→	see unary operators table
TcallvarF f	$13f$	funarg defined f frames outside call
Tcopyarg	35	copy reference to heap object onto the stack
Tcopyinput n	$4{+}74n$	copy n input values
Tcreateframe	72	creating a new frame
Tgoto	3	unconditional jump
Tiffalse	30	conditional jump, not taken
Tiftrue	30	conditional jump, taken
Tindex	89	vector indexing
Tmakevar	35	making a variable in let expression
Tmatchavailset n	$6{+}7n$	matching availability of n wires
Tmatchbool	35	matching a boolean value
Tmatchchar	35	matching a char value
Tmatchcon	43	matching a constructed type
Tmatchedrule	10	proceeding with the matchedrule
Tmatchfloat	35	matching a float value
Tmatchint	32	matching a integer value
Tmatchnone	11	matching none value ("*" or "_*")
Tmatchrule	20	matching a rule
Tmatchtuple	11	matching a tuple
Tmatchvar	36	matching a variable
Tmatchvector	11	matching a vector
Tmaybeconsume n	$47n$	consuming if available values on n wires
Tmkbool	85	making a boolean value
Tmkchain x	$371{+}94x$	adding a chain with x entries to closure
Tmkchar	84	making a char value
Tmkcon 0	112	making a constructor without arguments
Tmkcon n | $n{>}0$	$54n{+}107$	making a constructor node with n arguments
Tmkfloat	91	making a float value
Tmkfun p	$60p{+}166$	constructing a closure with p provided arguments
Tmkint	83	making an integer value
Tmknone	25	making an empty value
Tmktuple n	$52n{+}78$	making a tuple with n components
Tmkvector 0	81	constructing a vector of size 0
Tmkvector n | $n{>}0$	$52n{+}76$	constructing a vector of size n
Tpop	9	removing top of stack
Tpushvar 0	39	pushing a variable from current frame onto the stack
Tpushvar n | $n{>}0$	$11n{+}35$	pushing a variable from n frames outside onto the stack
Treturn n | $n{\leq}4$	116	return from function call, n number of return labels
Treturn n | $n{>}4$	$51{+}15n$	as above for more than four return labels
Tslide	53	sliding down the stack one position
TslidevarF 0	76	sliding down with units takes from variable
TslidevarF f | $f{>}0$	$79{+}11f$	as above, but variable stored f frames outside
Ttailcall a f	$22{+}36f{+}27a$	tail-recursive call with a arguments and frame difference f
TtoInt	2973	conversion of a float value into an int
TtoFloat	699	conversion of an int value into a float
Tunpack n	$44n{+}51$	unpacking a constructed type with n parts on the stack
Tunwindchain p	$49{+}51p$	unwinding a closure chain with p provided arguments
Tunwindfun p f	$301{+}51p{+}15f$	Tunwindchain plus function selection from f alternatives
Tconsumeset n	$32n$	Consume a set of n elements

A.1 Unary Operators

operator	cycles	comment
bool operand		
`not`	118	logical negation
int operand		
`-`	95	int unary minus
float operand		
`-`	97	float unary minus
`sqrt`	5177	square root
`sin`	11744	sine function
`cos`	11595	cosine function
`atan2`	14339	inverse tangens extended

A.2 Binary Operators

A.2.1 On Bool Values

operator	cycles	comment
`&&`	153	logical and
`||`	156	logical or
`=`	320	equality on booleans

A.2.2 Operators on numerical values

operator	cycles for int type	cycles for float type	comment
combination operators			
`+`	116	934	addition
`-`	116	938	subtraction
`*`	124	356	multiplication
`/`	206	962	division
`%`	1294	–	integral division remainder
comparison operators			
`<`	124	209	less than
`<=`	122	211	less or equal
`>`	124	209	greater than
`>=`	122	211	greater or equal
`=`	120	–	equal