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

Specific Targeted Research Project (STReP)
FET Open

D15 (WP3b): Prototype Implementation of time analysis

Due date of deliverable: 30th November 2006
Actual submission date: 28th February 2007

Start date of project: 1st March 2005 Duration: 36 months

Lead contractor: St Andrews University Revision: 1.4

Purpose: To describe the implementation of the worst-case execution time analysis that is formally defined in Deliverable D14.

Results: We have adapted the fully automatic prototype implementation of our Stack- and Heap-space analysis for HUME to analyse the worst-case execution time of HUME programs in relation to their input size.

Conclusion: Successfully accomplishing this deliverable has enabled us to test our WCET analysis against a number of sample HUME programs. These have, in turn, deepened our understanding of these nature and quality of the analyses.

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)	

Prototype Implementation of time analysis

Author: Steffen Jost <jost@cs.st-andrews.ac.uk>

Abstract

This document describes how to use the prototype implementation of the Worst-Case Execution Time analysis described in Deliverable D14 (WP3). The WCET analysis has been integrated into a single software package with the prototype implementations of the Heap- and Stack-space analyses (described in Deliverable D13). We therefore expect the reader to be familiar with Deliverable D13, including only the specifics for the WCET analysis in this document.

1 The parameterised WCET analysis

We have implemented the Worst-Case Execution Time (WCET) analysis described in Deliverable D14. We have created a stand-alone analysis tool, which processes programs written in Core-HUME, writes the constructed set of linear constraints in a human-readable format to a file, calls a standard linear programming solver (*lp-solve* [1]) to solve the constraints and returns the interpreted solution in text form. Our implementation was written in Haskell, a modern, powerful functional language [2]. This both has productivity benefits, reduces code maintenance costs, and allows straightforward integration with other HUME tools. It exploits commonalities with the implementation of the Stack- and Heap-space analysis described in Deliverable D13, allowing all three analyses to be delivered as part of a single software package.

Information on how to use the WCET analysis is given in Section 2. The result of the analysis is a bound on the number of processor cycles required to run the source HUME code on a Renesas M32C/85 processor. These bounds are linear in the size of the input for a program, if such linear bounds exist. In this document we will focus on the specific requirements of incorporating the WCET analysis into our general analysis strategy.

1.1 Amortised Analysis for Worst-Case Execution Time

Our general amortised analysis approach is described in Deliverable D13. While this does not depend, in principle, on the actual resource that is to be analysed, specifying only how *potential* can be embedded within annotated HUME types, the actual potential that is consumed or released by each language construct does, of course, depend on the specific resource that is being analysed. Different resources may also come into effect at different points in the derivation process. The implementation therefore uses a specific Haskell datatype (Figure 1) that represents all possible positions in the typing derivation where potential might be altered, including a number of context-dependent parameters. Adapting the implementation of our amortised analysis to a specific kind of resource then simply involves creating a mapping from this datatype to the actual resource consumption values. The general framework can be used for all kinds of resource.

Since dealing with time costs is somewhat more demanding than for space costs, the WCET analysis of Deliverable D14 therefore uses its own set of (possibly parameterised) timing constants ($\texttt{Tmkint}, \texttt{Tmkfun}(\cdot,\cdot), \texttt{Tcreateframe}, \texttt{Tgoto}, \ldots$). Unfortunately, these constants do not correspond exactly with the parameterised general resource cost parameters shown in Figure 1, since they may appear in different combinations at several places in the annotated type rules of Deliverable D14. We have therefore introduced a second datatype that exactly models these time constants (Figure 3). We can then define a general mapping from the parameterised general resource costs of Figure reffig:resourcedatatype to these time constants, as shown in Figure 2. This allows us to easily expand our implementation of the WCET analysis to processors other than the Renesas M32, once the number of cycles required for each HAM instruction is known.

Note that in Figure 2, some of the general cost parameters are mapped to the value zero, since they are not used for the WCET. A similar observation can be made for the Stack- and Heap-space analyses. In order to avoid solving unnecessary constraints, our implementation identifies zero cost parameters, and avoids generating constraints in these cases. This is one of the main reasons why a different count of constraints and resource variables can occur when the same program is analysed for different kinds of resource. Figure 4 liststhe program examples discussed in Deliverable D14 and the number of constraints and resource variables generated by our implementation of the WCET analysis Figures 5 and 6 show the analogous results for applying the Stack- and Heap-space analysis given in Deliverable D13, repeated here for convenience.

```
data ResPos = PBefore | PAfter                    -- Non-terminal rules with one premise,
 deriving (Eq,Show)                               --   cost effect before/after premises

data Res3Pos = P3Before | P3Middle | P3After      -- Non-terminal rules with multiple
 deriving (Eq,Show)                               --   premises


data ResConst =                                   -- Parameters for all cost kinds:

    RCpvar                                        -- pushing a variable onto the stack
  | RCmk    Type Value                            -- making a value of a certain type
  | RCuop   UnaryOp                               -- primitive unary operator
  | RCbop   BinaryOp                              -- primitive binary operator
  | RCcon   ConInfo Bool (Maybe Diamond)          -- making constructor
  | RCclcr  Bool [Type]                           -- creating closures storing args.
  | RCcloh  Bool [Type] ResPos                    -- overhead for executing a closure
  | RCapp   Bool Identifier [Type] ResPos AppKind -- function application
  | RClet   LetKind Int Res3Pos                   -- cost of a multiple let-binding
  | RClvar  LetKind Type                          -- cost of each let-binding
  | RCif    Bool ResPos                           -- conditional rule
  | RCmatch CaseKind ResPos                       -- general match rule cost
  | RCmrule CaseKind Res3Pos                      -- cost for individual match rule
  | RCpattern Pattern (Maybe ResPos)              -- cost to attempt a pattern match
 deriving (Eq,Show)
```

For example, the conditional allows four points where potential may be released or consumed: before and after evaluating the then- and the else-branch. Therefore `RCif` has two boolean parameters, one determining the branch and the other determining the position.

Figure 1: Haskell Datatype for resource independent cost parameters

```
costsTime :: TimeTable -> ResConst -> ResUnit    -- TIME for arbitrary machine models
costsTime tt RCpvar                 = tt Tpushvar
costsTime tt (RCmk UnitTyp _)       = tt Tmkunit
costsTime tt (RCmk BoolTyp _)       = tt Tmkbool
costsTime tt (RCmk IntTyp _)        = tt Tmkint
costsTime tt (RCmk FloatTyp _)      = tt Tmkfloat
costsTime tt (RCmk CharTyp _)       = tt Tmkchar
costsTime tt (RCmk _ (StringVal s)) = tt (Tmkstring (length s))
                                          ⋮
costsTime tt (RCif True  PBefore)   = tt Tiftrue
costsTime tt (RCif True  PAfter)    = 0
costsTime tt (RCif False PBefore)   = tt Tiffalse
costsTime tt (RCif False PAfter)    = tt Tgoto
                                          ⋮
```

The function `tt` is mapped to `ttM32` shown in figure 3, if option `-RKTimeM32` was specified.

Figure 2: Excerpt of the mapping from general cost parameters to the time constants

```
ttM32 :: TConstants -> ResUnit
ttM32 Tpushvar                = 30
ttM32 Tmkbool                 = 70
ttM32 Tmkint                  = 65
ttM32 Tmkfloat                = 65
ttM32 Tmkchar                 = 70
ttM32 (Tmkstring n)           = 13 * n + 140
ttM32 (Tmkcon n)              = 41 * n + 66
                              ⋮
```

Figure 3: Excerpt of the definition of time constants for Renesas M32

	Lines of Code		**WCET**	
Program Example	**HUME**	**Core-HUME**	**Constraints**	**Variables**
Factorial	10	23	24	29
List Summation	14	21	19	23
Iterated Multiplication	37	64	187	199
Drinks Vending Machine	64	138	681	540
Canny Edge Core Code	53	78	176	193

Figure 4: Number of constraints and resource variables generated by the WCET analysis

	Lines of Code		**Stack-space analysis**	
Program Example	**HUME**	**Core-HUME**	**Constraints**	**Variables**
Factorial	10	23	18	23
List Summation	14	21	12	16
Iterated Multiplication	37	64	142	154
Drinks Vending Machine	64	138	546	435
Canny Edge Core Code	53	78	134	151

Figure 5: Number of constraints and resource variables generated by the Stack-space analysis

	Lines of Code		**Heap-space analysis**	
Program Example	**HUME**	**Core-HUME**	**Constraints**	**Variables**
Factorial	10	23	16	21
List Summation	14	21	10	14
Iterated Multiplication	37	64	118	130
Drinks Vending Machine	64	138	480	373
Canny Edge Core Code	53	78	100	127

Figure 6: Number of constraints and resource variables generated by the Heap-space analysis

2 Usage

The analysis is invoked under Unix by `art3 [options] [infile]`. In order to force WCET analysis, the `-T` or `-RKTime` option must be specified. Options are provided to allow the derivation of time metrics for alternative processor types: currently only `-RKTimeM32` is supported, giving times in clock cycles for the Renesas M32C/85 processor. Further analysis options are as described for the space analysis implementation in Deliverable D13.

`-S, -RKStack` Performs Stack-space usage analysis.

`-H, -RKHeap` Performs Heap-space usage analysis [the default analysis].

`-T, -RKTime` Performs Worst-Case Execution Time analysis using the *default* time metric (equivalent to `-RKTimeM32`).

`-RKTimeM32` Performs Worst-Case Execution Time analysis in terms of clock cycles required for the Renesas M32C/85 processor.

`-nfp` Disables the experimental feature of assigning a potential to floating-point types (Section **??**).

`-v, -V` Verbose output for the preprocessing stage (only), where `-V` is more verbose than `-v`. Note that information from the analysis is currently written to the file `DBGTRACE`.

`--version` Displays version information.

`-h, --help` Prints usage and version information, plus the list of known resource kinds.

If multiple resource types are specified, then only the last will actually be analysed.

References

[1] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence).
`http://lpsolve.sourceforge.net/5.5`.

[2] The Glasgow Haskell Compiler.
`http://haskell.org/ghc`.