Automatic Amortised Worst-Case Execution Time Analysis

Christoph A. Herrmann* Armelle Bonenfant† Kevin Hammond*
Steffen Jost* Hans-Wolfgang Loidl‡ Robert Pointon§

Abstract

Our research focuses on formally bounded WCET analysis, where we aim to provide absolute guarantees on execution time bounds. In this paper, we describe how *amortisation* can be used to improve the quality of the results that are obtained from a fully-automatic and formally guaranteed WCET analysis, by delivering analysis results that are parameterised on specific input patterns and which take account of relations between these patterns. We have implemented our approach to give a tool that is capable of predicting execution costs for a typical embedded system development platform, a Renesas board with a Renesas M32C/85U processor. We show that not only is the amortised approach applicable in theory, but that it can be applied automatically to yield good WCET results.

1 Introduction

Worst-case execution time (WCET) analysis is required for a variety of embedded systems applications, especially those with safety- or mission-critical aspects. Common examples include avionics software and autonomous vehicle control systems [14]. Our work aims to construct *fully automatic* source-level static WCET analyses, that are correlated to actual execution costs. Since we must provide formal, automatically-produced *guarantees* on WCET bounds, we base our work on a high-quality abstract interpretation approach (AbsInt GmbH's **aiT** tool [5]), to give low-level timing information for bytecode instructions. We combine this with an equally formal, *type-based* approach that lifts this information to higher-level language constructs so that it can be applied to source programs. The problem is to maintain the strong WCET guarantees we need, while giving good quality information. In this paper, we consider a new approach to constructing WCET analyses, based on the idea of *amortisation* [16]. This represents the first attempt of which we are aware to provide an automatic amortised WCET analysis. We have produced a prototype implementation using our approach, and we report here on some preliminary results obtained using this analysis tool.

*School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SX.
email: {ch,kh,jost}@cs.st-and.ac.uk.
†IRIT, Université Paul Sabatier, Toulouse, France.
email: bonenfant@irit.fr
‡Ludwig-Maximilians Universität, München.
email: hwloidl@informatik.uni-muenchen.de
§Heriot-Watt University, Edinburgh.
email: rpointon@macs.hw.ac.uk

2 Amortised Time Analysis

Amortised cost approaches [3] allow costs to be averaged according to use. The basic intuition is that by amortising over the time costs incurred by common usage patterns (e.g. that for a stack, every *pop* is balanced by a *push*), we can construct timings that reflect more accurately real worst-case times. Typically, amortised analysis is performed by hand to determine the complexity of programs that involve complex data structures [13]. We have previously, however, applied the approach to give automatically derived, and provably correct, upper bounds on space costs for heap allocations [10]. In both cases, since alternative program execution paths may have very different costs, by amortising over common patterns we can avoid the needless over-estimation that would otherwise occur.

In this paper, we consider how the same approach can be applied to WCET. Our thesis is that such an approach can potentially reduce over-estimation *without losing formal guarantees that the analysis yields a genuine WCET.* We will show below that our amortisation-based WCET analysis can obtain WCET bounds that are close to the actual execution time. We first consider a simple example to illustrate the principles of our approach.

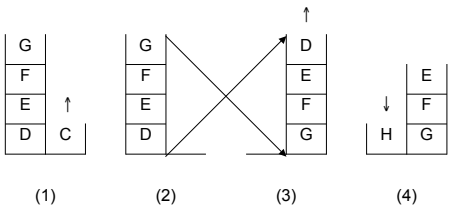

Figure 1: Queue implemented by two stacks

2.1 Example: Implementation of a Queue by Two Stacks

An example that is often used when teaching data structure abstraction and encapsulation is the implementation of a queue as two stacks. We will use this here to illustrate our amortised approach. Figure 1 shows a sequence of four queue configurations, in each of which the left stack is used to enqueue items and the right one to dequeue them. A queue abstraction can be built directly in terms of *push* and *pop* operations on the underlying stacks without needing to break the stack abstraction.

Enqueuing to a stack and dequeuing from a non-empty stack takes constant time. When the dequeuing stack becomes empty after popping C in (1), the contents of the enqueuing stack in (2) are popped in turn, and pushed onto the dequeuing stack as shown in (3). This then allows element D to be dequeued. Finally, new elements can be enqueued as shown in (4).

Reversing the stack by elementwise copying ((2) → (3)), however, takes time proportional to the number of items on the stack.

There are thus two worst cases. Firstly, the stack could become as large as the total number of elements, n, if the final element is enqueued before the first one is dequeued. The cost for the reversal would then be proportional to n. Secondly, each dequeued element could require a stack reversal if it is dequeued before the next one is enqueued. This would mean n stack reversals in total. Combining both worst cases would yield a worst-case estimation proportional to n^2. *However, both cases cannot occur at the same time*: as there are more stack reversals, the sizes of the stacks to be reversed decrease in size.

Amortised analysis treats these two extreme worst-cases quantitatively: we distribute the cost for the stack reversal equally among all the elements. The costs incurred by each element are: (1) a push for enqueuing; in the stack reversal (2) a test and (3) a pop followed by (4) a push; and for dequeuing (5) a test and (6) a pop. To process all elements, it follows that $6n$ basic stack operations are required.

By considering the queue structure as a whole rather than the operations of its components (stacks) in isolation, we have been able to amortise the worst-case cost, and *reduce it from a quadratic cost to a linear cost*. We can exploit this in an automatic analysis by using a model in which a *potential* [16] is given to each kind of data element. This potential represents the amortised worst-case cost per element of the structure. In the model (but not, of course, in an actual execution!), one unit of potential is released for each operation on an element. The potential must be sufficient to cover all operations. In our example, if we take into account only the stack operations, the potential would be 6 for each element.

2.2 Type-Based WCET Analysis

Our approach [11] is to analyse *source-level* constructs by building on standard type-checking algorithms. By doing this, we are able to gain information about programming constructs that may be lost through a compilation process, and so to obtain a better quality of analysis. Types also allow us to construct a *compositional* analysis, where functions/expressions/modules can be analysed independently. The flip-side is that we must have the program source available at analysis time. However, because we have a compositional analysis, previously obtained analysis information can be attached in the form of *meta-data* to binary programs or library code, and this will reveal nothing about the source implementation, apart from its WCET. In order to analyse source functions that use this code, it is not necessary to be able to *read* the binary file, or even to possess it, but only to see the meta-data.

Our work is undertaken in the context of the domain-specific programming language Hume, which embeds purely functional expressions in a powerful automaton-based process notation [9, 8]. We have constructed formally-correct type-based automatic analyses for determining worst-

case execution-time costs, based on the amortised cost approach described above [11], and implemented our rules in a prototype implementation (available from http://www.embounded.org). We will show some results that can be obtained from our analysis below.

The analysis works as follows: each construct in the source program is given a type using a normal type-inference algorithm. At the same time, the usage of potential is calculated for that expression. (Internal) cost variables are automatically associated with each (sub-)expression, and the analysis will generate a set of constraints over those variables that give an upper bound on the WCET. The constraint set is solved using a linear equation solver, and concrete cost solutions are mapped back to the source program. In this way, WCET costs are associated with each expression and each function that is used in the program.

Note that, while Hume deliberately simplifies the problem of constructing cost models and analyses, in order to allow us to focus on key research questions, rather than worrying about specific complexities, of e.g. C programming, the methods we are developing are, in fact, generally applicable. Hofmann and Jost have shown, for example, how a formally bounded heap analysis could be applied in an object-oriented setting such as Java [10]. Note also that, although our focus is on formally guaranteed WCET, and we have therefore built on abstract interpretation and types, amortisation can also, in principle, apply to other WCET approaches such as probabilistic approaches [1].

3 Obtaining WCET Bounds for HAM Instructions

The problem now is one of obtaining reliable WCET information for simple expressions so that costs for complex expressions can be constructed from the type-based analysis. We do this by first introducing an abstract machine (the Hume Abstract Machine or HAM [6]). We may then systematically determine WCET costs for each HAM instruction. Having determined the cost of each HAM instruction for a given architecture, and knowing how the Hume compiler will translate expressions to HAM instructions, we are able to analyse Hume programs for that architecture without needing to perform any compilation (even to HAM code), or any further analysis on the target machine. This yields a flexible and highly portable analysis.

The approach can yield an acceptable upper bound on WCET: we have shown in a previous paper [2] that composing costs of individual HAM instructions delivers a WCET result that can be within 2% of the cost of the WCET for a sequence of HAM instructions for the Renesas M32C/85U [4] architecture that we will also use in this paper. Processors such as the M32C, which have predictable time behaviour and low power consumption are especially of interest for use in embedded systems such as automotive applications. The processor therefore represents an important class of processor architectures that is employed in safety- and mission-critical systems.

3.1 Low-Level WCET Analysis

In this paper, we distinguish between *measured WCETs*, that are obtained in the obvious way, and *guaranteed WCETs*, that provide a formally guaranteed upper bound on WCET. Probabilistic approaches [1] extend the basic measurement approach by extrapolating from a set of measured WCETs to provide a probability function for the WCET.

Although it can be relatively simple to obtain time information by measurement, even for complex architectures such as a Pentium IV, this can be time-consuming, and it is not always straightforward to determine that the actual worst-case has been covered by the test input. Conversely, for guaranteed WCET obtained by static analysis, it is necessary to formalise the behaviour of the target processor. It can clearly be costly to construct such a model, especially where cache and pipeline effects are involved. For unpredictable architectures, such as the Pentium IV, even if it is feasible to model such a complex system, it may not be possible to construct a model that gives a tight WCET: the WCET could, in some cases, be one or two orders of magnitude greater than the typical execution time. However, once a model and analysis has been constructed, it can be applied repeatedly, and usually without significant programmer effort. A more detailed comparison of WCET approaches up to early 2007 can be found in the paper by Wilhelm et al. [17].

3.2 WCET Information for Individual HAM instructions

Our type-based approach can exploit information obtained using either measured or guaranteed WCETs for single HAM instructions. However, we can only formally *guarantee* upper bound times for source programs, if we use a correspondingly guaranteed approach to obtaining low-level WCET information. In this paper, we investigate both measured and guaranteed WCETs of HAM instructions for the Renesas M32C/85U, using machine code generated from HAM instructions by the IAR C compiler [15].

Guaranteed WCETs for each HAM instruction have been obtained using AbsInt GmbH's **aiT** tool [7]. This gives the WCET for each machine code fragment that is generated for a HAM instruction, using a static analysis approach that computes safe approximations for all possible cache and pipeline states that can occur at any given point in the program.

Measured WCETs for each HAM instruction have been obtained by repeated measurement of the instruction execution time, taking the worst case from 10000 runs, and using the cycle-accurate timer on the M32C to obtain software timings. To ensure accuracy, we use a "two-loop benchmarking" approach, which measures the time required for auxiliary measurement operations separately and subtracts this time from the measurement of interest. In this way, we avoid including measurement costs in the worst-case measurement. In order to guarantee that we measure the WCET for the program, we must, of course, provide an input which incurs the WCET during normal fault-free operation.

4 Example: Drilling Robot

We will now consider a slightly more in-depth example, to show how our amortised approach can be exploited to give WCET results and compare the outcome of the analysis with measured WCET times on the M32C processor.

4.1 Problem description

Our case study here is the simulation of a robot for drilling printed circuit boards. The robot can move a drilling head in fixed-sized increments (here represented as integers) in two dimensions.

```
pos_ok (xpos, ypos) = if xpos==0 && ypos==0
                      then 1 else 0;

step (xpos,ypos,actions,dps) =
 case actions of
    []     -> (dps, pos_ok (xpos, ypos))
  | (A:as) -> step (xpos,  ypos,   as,
                          ((xpos,ypos):dps))
  | (L:as) -> step (xpos-1,ypos,   as, dps)
  | (R:as) -> step (xpos+1,ypos,   as, dps)
  | (U:as) -> step (xpos,  ypos-1, as, dps)
  | (D:as) -> step (xpos,  ypos+1, as, dps);
```

Figure 2: Hume code for Drilling Robot

At each position, the drilling head can perform a drilling action. After all holes have been drilled, the drilling head is to be moved to its starting point. A similar application example would be a camera that can be turned around two axes and can take photo shots at particular orientations.

The robot can perform five operations: `A` is the drilling action; the other actions move the head by one position: `L` leftwards, `R` rightwards, `U` up and `D` down. Operations are described by the user-defined data type `OP` in Hume:

```
data OP = A | L | R | U | D
```

For example, if the action list is the four-element list `R:(D:(A:(L:(U:[]))))` (where `[]` represents the empty list and `(U:[])` constructs the one element list with `U` at the start of the list and an empty remainder), and the robot starts at position `(0,0)`, it will move right to `(1,0)`, down to `(1,1)`, drill, move left to `(0,1)` and finally up to `(0,0)`, the starting position.

The Hume function `step` (Figure 2) simulates each operation carried out by the drilling robot*. `step` takes four arguments: the current X- and Y-positions of the (`xpos` and `ypos`); the list of remaining actions to perform (`actions`); and an accumulating list that records the positions at which a drilling action has happened (`dps`). We perform case discrimation on the list of actions, where the list may be either empty, i.e. `[]`, or else a non-empty list whose first element is some action (`A`, `L`, ...), and whose remainder is the list of unprocessed actions (`as`). The second case is shown as the pattern `(A:as)` etc., which deconstructs the list with `A` as the first element,

*Of course, it would not be a major exercise to change this definition to actually drill a board.

case	`[]`	`(A:)`	`(L:)`	`(R:)`	`(U:)`	`(D:)`
measured	2512	1563	1533	1728	1932	2137
manual	2970	1891	1855	2109	2363	2617
/measured	1.182	1.210	1.210	1.220	1.223	1.225
automatic	3075	2033	2006	2269	2532	2795
/manual	1.035	1.075	1.081	1.076	1.072	1.068
/measured	1.224	1.301	1.309	1.313	1.311	1.308

Table 1: WCET Measurements and Analysis for each Case

binding the variable `as` to the rest of the list. The action list is processed element by element. There are three basic cases:

`[]` – the action list has been completely processed – we return the accumulated list of drilling positions, `dps`, paired with a control flag (calculated using the `pos_ok` function) that indicates whether or not the robot has returned to its initial position;

`A:as` – the current action is a drilling operation – having done this, we move on to the next action by calling `step` recursively on `as`, using the `((xpos,ypos):dps))` expression to place the current position, `(xpos,ypos)`, at the front of the list of drilling positions, `dps`;

`L:as, R:as, U:as, D:as` – the current action is a move – either `xpos` or `ypos` is changed as required, and we move on to consider the rest of the actions (`as`) recursively.

In the example above, there will be five recursive calls to `step`[†], starting with the `R` case, and ending with the `[]` case. The result will be the list of drilling positions, `((1,1):[])`, paired with an indicator that the final position is OK.

4.2 Amortised WCET Analysis

As in the queue example, amortisation allows us to avoid calculating the worst case cost of a list of actions as the length of the list multiplied by the worst case cost of any individual action. Our amortised analysis gives a bound for the worst-case execution time T_{WCET} in clock cycles depending on the number of occurences $\#X$ of each action constructor X in the input, which is $T_{\mathrm{WCET}} \leq 3075 + 2033 * \#\mathtt{A} + 2006 * \#\mathtt{L} + 2269 * \#\mathtt{R} + 2532 * \#\mathtt{U} + 2795 * \#\mathtt{D}$. This information is even more precise than expressing the WCET in terms of the list length, but if we wanted to reduce the information to this form, we could assume that $\#\mathtt{U} = \#\mathtt{D}$ when the robot returns to its initial position, i.e., $T_{\mathrm{WCET}} \leq 3075 + \lfloor 2663.5 * n \rfloor$ for a list of length n.

Table 1 compares the bound for each constructor with (a) the corresponding bound obtained from a manual analysis of a trace of the function `step`, summing the costs of all HAM instructions; and (b) the measured WCET. The absolute values given in the table refer to times in terms of clock cycles on the Renesas M32C/85U processor. Even for the prototype implementation of our analysis, we achieve manual analysis results within 23% of the measured WCET and automatic analysis results within 8% of our manually obtained results.

5 Conclusions

Amortised WCET analysis offers a way to construct costs that abstract over individual operations, and also to specialise WCET costs for particular input cases. Using our amortised WCET analysis approach for source-code analysis combined with information from the AbsInt **aiT** tool for machine-code analysis, we have been able to produce guaranteed WCET bounds that are reasonably close to measured WCETs (and indeed average-case times) for a real embedded systems platform. We are now working on tightening the bounds on WCET by improving the output of the automatic analysis and the quality of the code that is generated for HAM instructions. We are also applying the analysis to larger examples, e.g. ones taken from image processing or autonomous vehicle control [12]. Finally, we are

[†]The compiler will actually optimise these to be tail-recursive; effectively introducing a `goto` analogously to the implementation of an iterative loop in C, so there is no significant cost associated with this recursion.

working on exploiting high-level control flow information to guide the **aiT** analysis of the low-level compiled code, and so to obtain improved bounds.

We have explained our approach in terms of Hume, and have developed our initial analyses in the same context. Hume is a research notation that allows us to focus on core cost issues without distraction by many of the features that are found in production languages. However, now that the fundamentals have been properly worked out in this setting, the general static analysis techniques used here could, *in principle*, be applied to any other high-level language, including C. Moreover, the principle of amortisation could be applied to many forms of WCET analysis, not just the guaranteed WCET analysis we have described here.

Acknowledgements

This work is generously supported by EPSRC grant EP/C001346/1, by EU Framework VI IST-510255 (EmBounded) under the FET-Open programme, by an SOED support fellowship from the Royal Society of Edinburgh, and by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

References

[1] G. Bernat, A. Colin, and S.M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proc. 23rd IEEE RTSS 2002*, Austin, TX. (USA), December 2002.

[2] A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Proc. IFL 2006*, LNCS 4449. Springer-Verlag, 2007. To appear.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] Renesas Corp. http://www.renesas.com. 2007.

[5] AbsInt GmbH. http://www.absint.com. 2007.

[6] K. Hammond. Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In *Proc. First Central European Summer School, CEFP 2005*, LNCS 4164, pages 100–134. Springer-Verlag, 2006.

[7] K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hofmann, S. Jost, H.-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot, and A. Wallace. Towards formally verifiable resource bounds for real-time embedded systems. *ACM SIGBED Review— Special issues, ITCES06*, 3(4):27–36, October 2006.

[8] K. Hammond, G. Michaelson, and P.B. Vasconcelos. Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach. *ACM TOSEM*, 2007, under revision.

[9] K. Hammond and G.J. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Intl. Conf. on Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pages 37–56. Springer-Verlag, 2003.

[10] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *ESOP 2006*, LNCS 3924, pages 22–37. Springer-Verlag, 2006.

[11] S. Jost, H.-W. Loidl, K. Hammond, M. Hofmann, and A. Bonenfant. Generic amortised resource analysis for higher-order functional programs. In Preparation, 2007.

[12] G. Michaelson, A. Wallace, K. Hammond, I. Wallace, A. Bonenfant, and Z. Chen. Towards resource certified image processing software. In *SEAS DTC Annual Technical Conference, July 2006, Edinburgh, Conference Proceedings*, page A15. UK MoD, 2006.

[13] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[14] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. 5th Intl Workshop on WCET Analysis*, pages 21–24, 2005.

[15] IAR Systems. http://www.iar.com. 2007.

[16] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[17] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. Accepted for ACM TECS, 2007.