Worst-Case Execution Time Analysis through Types

Steffen Jost*, Hans-Wolfgang Loidl†, Norman Scaife‡, Kevin Hammond*, Greg Michaelson‡ and Martin Hofmann†
* *St Andrews University, St Andrews, UK. Email: {jost,kh}@cs.st-andrews.ac.uk*
† *Ludwig-Maximilians University, Munich, Germany. Email: {hwloidl,mhofmann}@tcs.ifi.lmu.de*
‡ *Heriot-Watt University, Edinburgh, Scotland. Email: G.Michaelson@hw.ac.uk*

***Abstract*—We construct a fully automatic static WCET analysis suitable for real-time embedded systems applications by combining a high-level static heap-space analysis technique with a machine-level worst-case execution time tool. We evaluate this approach by studying two typical and realistic real-time control applications, using a readily available commercial microcontroller.**

***Keywords*-WCET; analysis; static; type system;**

I. Introduction

Information about worst-case execution time (WCET) is critically important for many real-time systems. For example, we may require guarantees that a control loop is always fast enough to respond to its inputs. While testing a program might give some insight in its execution time behaviour, it cannot provide a guaranteed bound on execution time under all possible circumstances.

In this paper we present a source-level static analysis that determines a guaranteed upper bound on WCET, without running or altering the program. Determining the WCET of a particular source construct is difficult, since it may vary significantly depending on the overall machine state. For example, the WCET of a conditional expression depends on the branch that is taken. It is, however, generally infeasible to exhaustively examine all possible machine states at a given point within a program. Even worse, the WCET of a construct such as a function call can, in some cases, be arbitrarily large.

The only solution to this problem is to abstract all possible states into a smaller, more manageable classes of equivalent states, thereby trading precision for feasibility. If we manage to ensure that this loss of precision only happens in a safe way, e.g. by occasionally over-estimating, but never under-estimating costs, we can then establish a safe upper bound on the WCET. The approach that we take is especially radical, since we abstract the *entire* state and represent it by a single, non-negative rational number, the *"potential"* of the machine state. Note that we will never actually compute this number for any actual machine state other than the initial state. Instead, we examine the effect of each operation on the overall potential and define the *amortised cost* of an instruction $\mathcal{I}$ as a suitable constant such that

$$\textit{amortised cost}_{\mathcal{I}} \geq \text{actual cost}_{\mathcal{I}} - \text{potential before}_{\mathcal{I}} + \text{potential after}_{\mathcal{I}}$$

holds for all possible states (equality is preferred). The benefit is that the amortised cost is constant and does not depend on the machine state. The actual cost of an entire sequence of instructions is then bounded by the sum of the amortised costs plus the potential of the initial state.

In complexity theory this is termed "Amortised Analysis" [1]. A significant challenge of this technique is designing the abstraction of the machine state. Hofmann & Jost [2], [3] solved this problem for heap-space consumption, and produced an automatic inference, at the expense of restricting the potential to depend linearly on the structure of objects within a machine's memory. In this paper, we extend this technique to cover WCET analysis, covering two simple real-time control examples. An interesting improvement in the way we apply the amortised analysis technique is that the potential contributed by each memory object is assigned on a per-reference basis. Rather than counting references, we simply consider the point where aliasing is introduced. This allows us to assign differing potential to states which have many memory objects in common, but which differ slightly. The assignment of potential on a per-reference basis is a major improvement over previous work, such as that by Okasaki [4].

II. Towards an automatic static WCET analysis

We have adapted our previous amortised analysis for space usage to produce an automatic WCET analysis. We first construct a standard *typing derivation*, that assigns simple types to each source expression/function definition. Next, we define the amount of potential credited by each data object by annotating the type of each object with variables ranging over the non-negative rational numbers. We then generate a set of constraints over those variables, according to the dataflow and the actual cost occurring in each possible path of computation. This constraint set is then solved by a standard linear programming (LP-)solver, yielding concrete values for the type annotations. The annotated types then yield a linear closed-form expression depending on the input sizes, which represents an upper bound on the WCET. Since we have used a formal type-based approach, we could *formally prove* that the result of our analysis is *guaranteed upper bound* on WCET.

It is important to note this potential can be easily computed for large classes of the input data at once, since it amounts to calculating a linear closed-form expression over sizes. Hence it unnecessary to exhaustively consider all concrete inputs. For example, the analysis might tell

us that the WCET of a program depends only on the size of a specific input data structure, regardless of its content, and ignoring other inputs.

This analysis scales very well for two reasons. Firstly, it is *syntax-driven*: each instruction of the source program is examined precisely once. Loops in the source program are dealt with by identifying some variables contained in the constraint set. Secondly, the generated constraint sets are well-behaved and can easily be solved using a standard linear-programming (LP-)solver (we use **lp-solve** – http://lpsolve.sourceforge.net/5.5). Our constraint sets are several orders of magnitude below the size of problems that a standard LP-solver can be expected to handle nowadays.

Hume: While our static analysis approach is, of course, generally applicable, we need to concretise our approach in terms of some suitable language framework. We have therefore produced an implementation of our analysis for our experimental Hume language [5]. Hume fits our requirements for a testbed language well, since it has been designed to achieve *principled* and *predictable* functionality and behaviour, allowing us to focus on issues of cost modelling and analysis. Hume is a contemporary domain-specific programming language for real-time embedded systems targeting applications that require strong guarantees on resource bounds. The Hume design achieves this by separating an impoverished *coordination* layer, based on concurrent finite-state automata, from a rich *control* layer, based on a polymorphic functional language. The Hume programming model is based on the deployment of different Hume *levels* with different expressive properties. As discussed above, by augmenting a standard static type system with *amortisation*, it is possible to statically analyse the time and space requirements of Hume programs to different degrees of precision depending on the expressiveness of the Hume language level.

Experimental Setup: We use a simple Renesas M32C/85U microcontroller test board with an on-board USB interface. The system is programmed through an FoUSB debugger module which controls the board via the Flash programming socket. Several on-board switches are wired to the interrupt inputs and eight LEDs are connected to one of the generalised IO ports. The Hume compiler generates ANSI-compliant C code which is then compiled into machine code for the board using Renesas' M32C/80 C compiler (http://www.renesas.com, product ref. M3T-NC308WA). We obtain our basic instruction-level WCET information using AbsInt's **aiT** tool (http://www.absint.com). **aiT** uses *abstract interpretation* to provide a guaranteed, and tight, upper bound on actual run-times for C/machine-code fragments with known data inputs. Where relevant, it includes precise models of cache [6] and pipeline behaviours [7].

We exploit this *low-level* information for specific data inputs to give a *high-level* amortised analysis covering arbitrary inputs. As part of the compilation process, the Hume compiler produces an intermediate bytecode form. By applying **aiT** to the C code for each bytecode, we can obtain the corresponding WCET on the Renesas M32C/85U (or other processors). The amortised analysis technique is then applied to each Hume language construct, using WCET information for each bytecode block as its basic cost metric. The benefit of this approach is that the high-level analysis, using amortisation to track a highly abstracted state, can then relate WCET to source-level datatypes and language constructs, thereby dealing with complex programs and system states, including loops, conditionals and other information.

III. Experiments

A. PID Controller

Our first example is a simple 2-degree-of-freedom PID controller for controlling a classic "ball and beam" experiment, as shown in Figure 1. The experiment itself was simulated using the open-source dynamics engine "OpenODE" (http://www.ode.org) with a minimal environment based on *openGL*. This simulation is a fully-featured physical simulation of a ball and beam with actuator control and direct measurement of the beam angle and ball position.

$$\begin{aligned} P_k &= K_p * (e_k - e_{k-1}) \\ I_k &= Ki * T * e_k \\ D_k &= (K_d/T) * (e_k - 2 * e_{k-1} + e_{k-2}) \\ CV_k &= CV_{k-1} + P_k + I_k + D_k \\ e_k &= SP_k - PV_k \end{aligned}$$

Figure 1. Equations for a 2 degree-of-freedom PID controller with rectangular integration

Our Hume code (Figure 2) is a direct implementation of the recursive equations for a 2-DOF controller with rectangular integration, see Figure 1, with two additional actions for initialisation and resetting if the ball comes off the beam. The main control loop performs the integration using previous instances of loop variables to implement unit delays.

```
match
  (c,(false,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,onbeam)) ->
  (('x',vCVk1),*,(true,0.0,0.0,c))
|(c,(true,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,0)) ->
  (('2',vCVk1),(vSPk,vCVk1,vPVk,phi,0,c),(true,0.0,0.0,c))
|(c,(true,ek1,ek2,lc),(vSPk,vCVk1,vPVk,phi,1)) ->
 let vT = c -. lc in
 let ek = vPVk -. vSPk in
 let vPk = vKp *. (ek -. ek1) in
 let vIk = vKi *. vT *. ek in
 let vDk = ((vKd/.vT) *. (ek -. 2.0*.ek1 +. ek2)) in
 let vCVk = vCVk1 +. limitPID(vPk +. vIk +. vDk) in
  (('x',vCVk),(vSPk,vCVk1,vPVk,phi,1,vT),(true,ek,ek1,c))
```

Figure 2. Hume code for the 2-DOF PID controller

We have measured the runtime of this code on the Renesas M32C/85U micro-controller board for one thousand iterations. We found the runtime for the controlling loop to be within 27473 and 28406 clock cycles, with the average being 27742 clock cycles.

Our WCET analysis generates 459 constraints over 793 variables for this program. This is easily solved by any standard LP-solver, which can usually handle problems described by several hundred thousand constraints. The analysis gives an upper bound for the WCET of 36682 clock cycles. This is 29.1% above the worst runtime that we measured (this is, of course, not necessarily the actual WCET, since we may not have covered the actual worst case in our testing).

An interesting detail in the analysis' result is that we obtain an arbitrarily large weight for the value `False` in the first element of the third tuple of the result. An arbitrary potential occuring in the output of a program indicates that this case cannot occur at all, since it would allow us to justify an impossible WCET of zero. Examination of the code confirms this, since all three branches have the constant `True` at this position, so the value `False` cannot indeed occur. This *init-flag* is only used to signal the start (or resetting) of the PID controller, causing it to initialise itself. It follows that it is always set to `True`, once the controller has completed its first loop, and our analysis can detect this. It thus goes beyond what is possible for a typical basic WCET analysis.

B. Inverted Pendulum Example

Our next example is an inverted pendulum controller. This implements a simple, real-time control engineering problem. A pendulum is hinged upright at the end of a rotating arm. Both rotary joints are equipped with angular sensors, which are the inputs for the controller (arm angle θ and pendulum angle α). The controller should produce as its output the electric potential for the motor that rotates the arm in such a way that the pendulum remains in an upright position.

For this example we used real hardware, rather than the simulation approach we used for the previous example. The sensor outputs and the input to the motor may range in voltage between -10V and +10V. Since the Renesas M32C/85U only accepts voltages between 0V and 5V, three simple *operational amplifier* (op-amp) voltage-scaling circuits were used to interface the pendulum to the development board.

For control, we generate a state-space representation of the system:

$$\begin{aligned} \dot{\underline{X}} &= A_c\underline{X} + B_c\nu_m \\ y &= C_c\underline{X} \end{aligned}$$

by starting from the Lagrangian dynamics of the system followed by linearisation. The state vector is $\underline{X} = (\theta, \alpha, \dot{\theta}, \dot{\alpha})^T$ and the state matrix (A_c) and input/output (B_c/C_c) matrices can be computed from the known physical properties of the system such as the length and weight of the arm and constants which relate motor current to torque (ν_m is the voltage applied to the motor). The form of these continuous-time matrices is not interesting but we can enter them into Matlab or Octave and use the `c2d` function to discretise the representation for a predefined sample rate using standard techniques. Given suitable covariance matrices (experimental values) we can compute the optimal feedback gain K using the function `dlqr` from Matlab or Octave's control toolbox.

In order to implement the control loop. we choose the simple option of setting $\dot{\theta}$ and $\dot{\alpha}$ to zero and using optimal gain without any estimates of the angular velocities. Note that we actually use as output matrix $C_c = (0, 1, 0, 0)$ so we are only controlling α, not θ. This means that our implementation is prone to arm drifting despite the pendulum being controlled in the vertical position.

The Hume code comprises about 180 lines of code, which are translated into about eight hundred lines of intermediate code to be analysed. The automated analysis generates 1115 linear constraints over 2214 different variables. The overall runtime of the analysis itself was less than second ($\sim$0.67s) on a standard laptop (1.73Ghz Intel Pentium M processor with 2MB cache and 1GB of memory). This includes the time for solving the generated linear programming problem ($\sim$0.26s).

We have measured the best-case (36118), worst-case (47635). and average number of clock cycles (42222) required to process the controlling loop over 6000 iterations during an actual run, where the Renesas M32C/85U actually controlled the inverted pendulum. Compared to the WCET bound given by our automated analysis (63678) we have a margin of 33.7% between the predicted WCET and the worst measured run. The pendulum controller can only be made stable with a sample rate of less than about 10ms. The measured loop time is 1.488ms, while our predicted loop time would be 1.989ms, showing that our controller is guaranteed to be fast enough to successfully control the pendulum under all circumstances.

We note that the analysis performed poorly on the branches dealing with starting and stopping the experiment, for example the setup calibration using button presses. This happens because our prototype implementation of the analysis over-estimates the time required for processing matches on multiple patterns by applying an overly-pessimistic (but safe) assumption about match failure (essentially we ignore that some patterns may be irrefutable). Small computational branches that are dominated by the initial large pattern match thus produce a fairly high over-estimation of the WCET. The worst of these branches had a measured maximum runtime of 2669 clock cycles, where our analysis guaranteed a WCET of 7702 clock cycles. However, manually rearranging the order of the pattern matches could reduce the guaranteed WCET down to a very reasonable 2711 clock cycles for that branch.

IV. Related work

Our WCET analysis builds on previous work by Hofmann and Jost for a functional programming language [2] and a comprehensive subset of Java [3]. This work has been extended to stack-space analysis [8]. However, none of these approaches deals with worst-case execution time.

A system that combines a type-based approach with the automatic generation of checkable certificates for time bounded computation is described in [9], but this lacks any inference of the bounds.

Other functional notations with ad-hoc techniques for analysing resource consumption are GeHB and RT-FRP [10], with a two-level *staged* notation that also builds on the technique used by Hofmann and Jost.

Another important type-based approach to infer size bounds is that of *sized types* [11]. Several analyses on heap or stack space usage build on this concept (e.g. [12], [13], [14], [15]), but again none considers worst-case execution time. Vasconcelos' PhD thesis does claim, however, that WCET should be within reach of these methods.

Finally, a variety of academic and commercial tools exist for calculating guaranteed bounds on *worst-case execution time (WCET)* [16], including aiT[17], bound-T[18], SWEET[19]. However, all the tools of which we are aware suffer from the same restrictions as the **aiT** tool, namely intensive manual guidance by the programmer, and restriction to specific input values.

V. Conclusion

In summary, our results clearly show that our amortised analysis technique is capable of inferring good bounds for real-time applications. The bounds obtained by our prototype implementation for realistic embedded systems control applications, such as the ball-and-beam and inverted pendulum controllers, show a general over-prediction by about one-third over the maximum runtimes *that we have observed* for a range of data inputs. We deem this to be an acceptable margin for *automatically-generated and formally-guaranteed* WCET bounds.

References

[1] R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, no. 2, pp. 306–318, April 1985.

[2] M. Hofmann and S. Jost, "Static Prediction of Heap Space Usage for First-Order Functional Programs," in *Proc. POPL 2003: ACM Symp. on Principles of Programming Languages*. ACM, 2003, pp. 185–197.

[3] ——, "Type-based amortised heap-space analysis (for an object-oriented language)," in *Proc. ESOP 2006: European Symp. on Programming*, P. Sestoft, Ed. Springer LNCS 3924, 2006, pp. 22–37.

[4] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1998.

[5] K. Hammond and G. Michaelson, "Hume: a Domain-Specific Language for Real-Time Embedded Systems," in *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*. Springer-Verlag LNCS 2830, Sep. 2003, pp. 37–56.

[6] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt, "Cache behavior prediction by abstract interpretation," *Science of Computer Programming*, vol. 35, no. 2, pp. 163–189, 1999.

[7] M. Langenbach, S. Thesing, and R. Heckmann, "Pipeline modeling for timing analysis," in *SAS*, ser. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, 2002, pp. 294–309.

[8] B. Campbell, "Stack Usage Analysis," Ph.D. dissertation, Edinburgh University, 2008.

[9] K. Crary and S. Weirich, "Resource Bound Certification," in *Proc. ACM Symp. on Principles of Prog. Langs.*, 2000, pp. 184–198.

[10] Z. Wan, W. Taha, and P. Hudak, "Real-time FRP," in *Intl. Conf. on Functional Programming (ICFP '01)*. ACM, Sep. 2001.

[11] R. Hughes, L. Pareto, and A. Sabry, "Proving the Correctness of Reactive Systems Using Sized Types," in *Proc. POPL '96: ACM Symp. on Principles of Programming Languages*. St. Petersburg Beach, Florida: ACM, January 1996.

[12] R. Pena and C. Segura, "A First-Order Functl. Lang. for Reasoning about Heap Consumption," in *Draft Proc. Intl. Workshop on Impl. and Appl. of Functl. Langs. (IFL '04)*, 2004, pp. 64–80.

[13] W.-N. Chin and S.-C. Khoo, "Calculating Sized Types," *Higher-Order and Symbolic Computing*, vol. 14, no. 2,3, pp. 261–300, 2001.

[14] P. Vasconcelos and K. Hammond, "Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs," in *Proc. IFL '03: International Workshop on Implementation of Functional Languages*. Springer-Verlag LNCS, 2004, pp. 86–101.

[15] B. Grobauer, "Cost recurrences for DML programs," in *Proc. ICFP '01: Sixth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM Press, 2001, pp. 253–264.

[16] 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," 2008, Accepted for TECS.

[17] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *EMSOFT*. Springer-Verlag LNCS 2211, 2001, pp. 469–485.

[18] N. Holsti and S. Saarinen, "Status of the Bound-T WCET tool," in *Proc. WCET '02: Int'l Workshop on Worst-Case Execution Time Analysis*, June 19-21 2002.

[19] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz, "Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems," in *Proc. ISOLA '06: Int'l Symp. on Leveraging Applications of Formal Methods*, Paphos, Cyprus, November 2006.