Back Annotation in Action: from WCET Analysis to Source Code Verification

Vítor Rodrigues[1], Mário Florido[1], Simão Melo de Sousa[2]

[1] DCC-Faculty of Science & LIACC, University of Porto
vitor.gabriel.rodrigues@gmail.com
amf@ncc.up.pt
[2] DI-Beira Interior University & LIACC, University of Porto
desousa@di.ubi.pt

Abstract. One of the essential safety parameters of real-time programs is the respect for the temporal restrictions imposed by the real-time systems. The assessment of this requirement is commonly based on the computation and verification of the *worst-case execution time* (WCET). The WCET is the main parameter used in schedulability analysis and can be used both on software optimization and on hardware dimensioning. In this context, the availability of WCET information at source code level is highly desirable.
The main challenge is to establish a link based on annotations between the source code and the machine code compiled to a particular target architecture. At same time, the design of this annotation process in a compiler-independent way is also transparent and modular. The contributions on this paper report the *back-annotation* mechanism of the platform ACCEPT (Abstraction Carrying CodE Platform for Timing validation). In particular, we focus on the integration of a static analyzer used for WCET analysis into the software development environment.

1 Introduction

Nowadays, many distributed systems have adaptive configuration mechanisms that allow software actualizations after deployment. However, the auto-reconfiguration of such systems requires information that is transmitted across the network, fact that imposes new safety requirements on software actualizations. Therefore, software actualizations must be object of verification in respect to some safety policy. One important class of safety requirements is based on the *worst-case execution time* of an application [5]. The main challenge in the implementation of verification mechanisms based on WCET on embedded real-time systems is the efficiency in terms of computational resources.

Modern microprocessors have many specialized hardware features which increase the difficulty to estimate the WCET. In practice, this means that although real-time applications are written in high-level programming languages, the analysis of the WCET must be performed at hardware level. However, if the loss of abstraction resulting from compilation is definite, the feedback of the

WCET analysis results back to the source level is no longer possible. To prevent from this, the *standard* DWARF [13] provides compiler debug information with a bidirectional correspondence between the source code lines and the memory positions which hold the respective machine code. In Section 4, we present an example that shows how the DWARF standard is used by the ACCEPT's *back-annotation* mechanism.

The theoretical foundations for the WCET analysis are the theory of Abstract Interpretation (AI) combined with Integer Linear Programming (ILP) techniques [14]. The static analyzer is based on a uniform fixpoint semantics which is programming language-independent. To this purpose, a two-level denotational meta-language was developed in the light of [8], that we use to represent the semantics of programs in a uniform way. At the higher level of the meta-language are defined *meta*-programs that encode the control flow graph of the program which interpretation is always the same, regardless of the abstract domain. At the lower level of the meta-language are instantiated different abstract interpretations, under the form of denotational semantic functions, that are used to compute specific abstract properties, defined by a proper abstract domain.

The contribution of this paper is the integration of a back-annotation mechanism into the ACCEPT platform (see Section 4) considering the C programming language and the latest generation C compiler for the ARM target platform. A brief introduction to the uniform program semantics is given in Section 3. Conclusions and future work are discussed in Section 5.

2 Related Work

Falk *et al.* present in [6] a compilation process for C programs, designated by WCET-*aware* C *Compiler* (WCC), that incorporates the notion of WCET into the compiler and delegates the WCET analysis on the static analyzer aiT [1]. The advantage of using the aiT tool is the possibility to integrate state-of-art static analysis, namely the analysis of the pipeline behavior and cache structures, into the compiler environment. Although the ACCEPT's back-annotation mechanism is compiler-independent, its static analyzer also supports the advanced hardware features such as pipelines and cache memories. On the other hand, the main advantage of the ACCEPT's static analyzer is its adequacy to the low resource capabilities of embedded systems, specifically in terms of the necessary number of iterations required to achieve fixpoints.

Besides being a necessary component for the production of optimized code based on cost functions, the analysis of the WCET is by itself valuable for the programmer in those cases where the visualization of the WCET at source level is possible. In such a scenario, the analysis of the WCET is pragmatically performed at source level by allowing the programmer to abstract from the machine code details, also taking advantage from the fact that the WCET analysis is automatic. Along these lines, [7] presents a method for *loop unrolling* based on the WCC platform, which optimizes cycles by means of code expansion at the same time that explores maximal reduction of the WCET.

An alternative way to establish the bridge between the analyzed machine code and the high-level representation of source code is to use the DWARF debug information [13] generated by the compiler and included inside the executable binary. Despite the limitations that the DWARF standard revels when compiler optimizations are active, its use allows the integration of a generic compiler into the ACCEPT platform, at the same time that allows WCET data to become visible in the development environment. This approach is followed by Plazar *et al.* in [9], where is presented a variant of the WCC platform. The main difference between these two approaches is the granularity of the back-annotation mechanism. While with WCC the data about WCET are exported to the compiler's back-end, which holds an exact correspondence between the source code constructs and the machine code, the use of a generic compiler in the shape of a black-box makes the annotation of WCET data dependent from the DWARF internal representation, which only associates source code lines to the corresponding memory instruction addresses.

3 Program Semantics

Program semantics are expressed by the relational-algebraic constructs of the meta-language, regardless of the programming language being object of analysis. In complement, the particularities of some programming language are expressed by a denotational semantics which is used as a parameter by the relational algebra [10]. Pragmatically, this separation brings the possibility to derive a *meta*-program, composed by polymorphic relational operators, that reflects the structure of the program. Later, this meta-program is simulated using the denotational semantic functions as arguments. The implementation of this mechanism is based on the hierarchy of semantics proposed by Cousot in [4, Theorem 33] and on the two-level meta-language proposed by the Nielsons in [8].

As an example, consider the source code example in Figure 1 and the corresponding machine code in Figure 2(a) generated by GNU GCC compiler for the ARM target platform [3]. In [11], we demonstrate that the meta-semantic formalism defined at relational level can be effectively used to generate all possible program paths throughout a refinement process [4, Section 6], and that the same formalism supports interprocedural analysis according to the functional approach described in [12, Section 3].

```
int main(void) {
  int y = foo ();
  return y;
}

foo () {
  int x = 5;
  while (x>0) {
    x--;
  }
  return x;
}
```

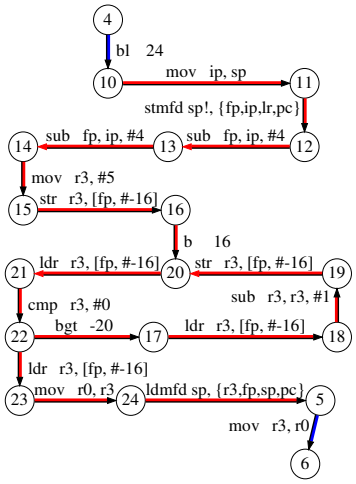
Fig. 1. Source Example

Figure 2(a) illustrates the form of the relational semantics. It is composed by a set of pairs of states, each state corresponding to a program transition. The order of the relational semantics is based on the intermediate states and corresponds necessarily to the order of the program's instructions. The assignment to an intermediate state is done according to a weak topological order [2]s. Simple states are numbered with the prefix "n" and the

states that introduce non-determinism (*heads*) are numbered with the prefix "h". Additionally, each state contains the procedure to which it belongs in "{}".

```
 n1{0}: mov ip, sp             :n0{0}  [ d1 ]
 n2{0}: stmfd sp!, fp,ip,lr,pc :n1{0}  [ d2 ]
 n3{0}: sub fp, ip, #4         :n2{0}  [ d3 ]
 n4{0}: sub sp, sp, #4         :n3{0}  [ d4 ]
n10{1}: bl 24                  :n4{0}  [ d5 ]
 n6{0}: mov r3, r0             :n5{0}  [ d6 ]
 n7{0}: str r3, [fp, #-16]     :n6{0}  [ d7 ]
 n8{0}: ldr r3, [fp, #-16]     :n7{0}  [ d8 ]
 n9{0}: mov r0, r3             :n8{0}  [ d9 ]
  HALT: ldmfd sp, r3,fp,sp,pc  :n9{0}  [ d10 ]
n11{1}: mov ip, sp             :n10{1} [ d11 ]
n12{1}: stmfd sp!, fp,ip,lr,pc :n11{1} [ d12 ]
n13{1}: sub fp, ip, #4         :n12{1} [ d13 ]
n14{1}: sub sp, sp, #4         :n13{1} [ d14 ]
n15{1}: mov r3, #5             :n14{1} [ d15 ]
n16{1}: str r3, [fp, #-16]     :n15{1} [ d16 ]
n20{1}: b 16                   :n16{1} [ d17 ]
n18{1}: ldr r3, [fp, #-16]     :n17{1} [ d18 ]
n19{1}: sub r3, r3, #1         :n18{1} [ d19 ]
n20{1}: str r3, [fp, #-16]     :n19{1} [ d20 ]
n21{1}: ldr r3, [fp, #-16]     :n20{1} [ d21 ]
n22{1}: cmp r3, #0             :n21{1} [ d22 ]
n17{1}: bgt -20                :h22{1} [ d23 ]
n23{1}: ldr r3, [fp, #-16]     :h22{1} [ d24 ]
n24{1}: mov r0, r3             :n23{1} [ d25 ]
 n5{0}: ldmfd sp, r3,fp,sp,pc  :n24{1} [ d26 ]
```

(a) Relational semantics



(b) Control Flow Graph

Fig. 2. Alternative Representations at Machine Code Level

4 Back-Annotation

Having the local execution times computed by the static analyzer and the WCET calculated by the ILP component, the back-annotation mechanism pushes the results of WCET analysis beyond the limits of the machine language. The objective is to annotate the maximal local execution times on each source code line and the overall WCET on the top level procedure. To this end, the back-annotation mechanism relies on a mapping between the program points in the source code and the memory addresses (*program counter*) where the corresponding machine code is stored. A human readable representation of this mapping is found in the debug section ELF ".debug_line" which can be obtained from the executable binary using the *dwarfdump* tool. For the example in Figure 3(a), the referred mapping is given by the two first columns in Figure 3(b).

To this matrix is added a third column that contains the labels of the states defining the relational semantics in Figure 2(a). These labels are obtained by post-processing the relational semantics in order to establish a link between each label and the corresponding source code line. In this way, the program counter addresses enable a correspondence between the source program and the control flow graph of the machine program and, therefore, provide the means

```
1  // wcet = Just 399.0 cpu cycles
2  int main(void){ // 0 cycles
3      int y = foo(); // 8 cycles
4      return y;    // 8 cycles
5  }
6
7  foo()
8  {
9      int x = 5;   // 8 cycles
10     while (x > 0){  // 6 cycles
11         x--;     // 10 cycles
12     }
13     return x;    // 10 cycles
14 }
```

(a) Annotated Source Code

Line	Program Counter	Machine Point
2	0x8450	n0
3	0x8460	n4
4	0x846c	n7
5	0x8470	n8
8	0x8478	n10
9	0x8488	n14
10	0x8490	n16
11	0x8494	n17
10	0x84a0	n20
13	0x84ac	h22
14	0x84b0	n23
14	0x84b8	n5

(b) Post-processing of DWARF Information

Fig. 3. Example of *Back-Annotation*

to annotate the source code with the local execution time bounds obtained by abstract analysis.

Analyzing the first column in Figure 3(b), one can see that to the same source code line may correspond more that one program counter. This repetition can be related to the beginning of a basic block, the beginning of an epilogue of a procedure or the end of a prologue of a procedure. In particular for the source code line 10, the first program counter specify the branch instruction that initiates the **while** cycle, whereas the second program counter specifies the first instruction of the **while** cycle. Analyzing the third column in Figure 3(b), it is demonstrated that the relational semantics in Figure 2(a) correctly expresses the control flow of the machine program. Considering the same example of the source code line 10, the relation **d17** specify precisely the transition between the labels **n16** and **n20**.

5 Conclusions and Future Work

In this paper we presented the back-annotation mechanism used in the ACCEPT platform to provide an effective WCET analysis at source code level. The annotation process does not depend on a particular compiler but on the implementation of DWARF standard by the compiler. Although the actual version of the GNU compiler for the target platform ARM does not yet support the latest DWARF format, one of the most relevant improvements in this standard is the description of optimized code. In the short term, this fact will enable the support for compiler optimization in a transparent way. On the other hand, the correspondence between the program points in the source code to the program points in the machine code also enables the verification of the correction of the compilation using the theory of abstract interpretation [11]. The next challenge is the extension of this mechanism to allow the back-annotation of machine level data, such as register values and memory locations, to the source level so that user-defined specifications can be verified.

6 Acknowledgments

This work is partially funded by LIACC, through the Programa de Financiamento Plurianual, FCT, and by the FAVAS project, PTDC/EIA-CCO/105034/2008, FCT.

References

1. AbsInt: (2011), http://www.absint.com/pag/
2. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: In Proceedings of the International Conference on Formal Methods in Programming and their Applications. pp. 128–141. Springer-Verlag (1993)
3. the GNU Compiler Collection, G.: (2011), http://gcc.gnu.org/
4. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Electronic Notes in Theoretical Computer Science 6 (1997)
5. Engblom, J., Ermedahl, A., Sjodin, M., Gustafsson, J., Hansson, H.: Worst-case execution-time analysis for embedded real-time systems. International Journal on Software Tools for Technology Transfer 4, 437–455 (2003)
6. Falk, H., Lokuciejewski, P., Theiling, H.: Design of a wcet-aware c compiler. In: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia. pp. 121–126. ESTMED '06, IEEE Computer Society, Washington, DC, USA (2006)
7. Lokuciejewski, P., Marwedel, P.: Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems. pp. 35–44. IEEE Computer Society, Washington, DC, USA (2009)
8. Nielson, H.R., Nielson, F.: Pragmatic aspects of two-level denotational metalanguages. In: Proceedings of the European Symposium on Programming. pp. 133–143. ESOP '86, Springer-Verlag, London, UK (1986)
9. Plazar, S., Lokuciejewski, P., Marwedel, P.: A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations. In: Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP. pp. 49–52. Barcelona / Spain (Dec 2008)
10. Rodrigues, V., Florido, M., Melo de Sousa, S.: A functional approach to worst-case execution time analysis. In: 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP) (July 2011)
11. Rodrigues, V., Florido, M., Melo de Sousa, S.: Towards adaptive real-time systems by worst-case execution time checking. Tech. rep., Artificial Intelligence and Computer Science Laboratory (LIACC)- University of Porto (May 2011)
12. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis, chap. 7, pp. 189–233. Prentice-Hall, Englewood Cliffs, NJ (1981)
13. The DWARF Debugging Standard : http://www.dwarfstd.org/
14. Wilhelm, R.: Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, vol. 2937, pp. 309–322. Springer Berlin (2003)