A constraint-based WCET computation framework

Hajer Herbegue, M Filali, Hugues Cassé

A Constraint-Based WCET Computation Framework

Hajer Herbegue Mamoun Filali Hugues Cassé

CNRS IRIT
Université de Toulouse, France
first_name.last_name@irit.fr

ABSTRACT

OTAWA is a tool dedicated to the WCET computation of critical real-time systems. The tool was enhanced in order to take into account modern micro-architecture features, through an ADL-based approach. Architecture constraints are expresses such that they can be solved by well known efficient constraint solvers. In this paper, we present how we could describe some complex architecture features using the Sim-nML language. We are also concerned by the validation and the animation point of views.

1. INTRODUCTION

OTAWA is a tool dedicated to the analysis of critical real-time systems. The worst case execution time (WCET) computation is one of the crucial analyses it provides. In a previous paper [12], we have shown how to enhance this tool in order to take into account modern micro-architecture features and complex instruction set architectures. An approach based on architecture description language (ADL) for the execution time analysis considers the architecture description in the Sim-nML language [11]. We have surveyed how to express architecture constraints such that they can be solved by well-known efficient solvers. In this paper, we present an extension of the Sim-nML language that allows to handle a set of complex features of modern processors. We are also concerned by a validation point of view. Since the validation of almost all currently used solvers is out of reach, we extended the OTAWA framework to allow the validation of the results returned by these solvers. This approach is not new: it is already used in assistant theorem provers [7, 16] where results provided by other less reliable but more efficient tools [2] are established again, i.e., validated. The inclusion of the validation layer is a novelty of OTAWA since it represents an easy and fast way to verify if analysis results meets the initial architecture specification.

The rest of our paper is organized as follows: Section 2 is an overview of the related works. Section 3 recalls the context of our work. We present the OTAWA tool and the constraint-based approach for WCET computation. Section 4 presents the Sim-nML language and its extension to describe some complex features of modern processors. Section 5 presents another contribution of the paper related to validation and animation aspects. Section 5 draws some conclusions.

2. RELATED WORKS

Over the last years, many studies have been undertaken with respect to WCET computation for pipeline architectures. Among them, we mention the work of [15] concerned by verifying structural properties related to the wellformedness of the architecture. A graph-based model of the processor is generated from an ADL-based description. Architecture structural properties are validated using algorithms applied on the graph. With respect to dynamic aspects, the model checking approach has been experimented with mitigated results [9]. The basic timed automata model enhanced with game theory aspects [4] seems promising. Nevertheless, abstract interpretation based approaches [19] are today well established. In this paper, we are interested in the AI based approach on constraint satisfaction [6] and its validation.

3. ADL-BASED APPROACH FOR WCET COMPUTATION

OTAWA [3] is a framework dedicated to WCET computation founded on an abstraction layer that describes the target hardware and the instruction set architecture (ISA), as well as the binary code under analysis. The instruction set architecture (ISA) specification is expressed in the Sim-nML language. The hardware architecture is described through an XML format. Yet only some architectures can be handled by such a model. A pipeline analysis consists in modeling the execution of basic blocks[1] on the pipeline and then computing the corresponding execution costs [18]. A constraint-based approach, presented in [12], is based on ADL processor descriptions and uses constraint specification languages and resolution methods to compute the time cost of a basic block. The target processor is described using the Sim-nML language [10, 17]. In addition to the ISA level, the architecture description includes the hardware components and the execution model of instructions. Sim-nML was extended to support such a description of the processor. The carried analysis aims to estimate the time cost of basic blocks of a program. Using the processor description in the Sim-nML language and the basic block, we generate a constraint-based description. The execution time of a basic block is described as a Constraint Satisfaction Problem (CSP) [6]. This approach handles complex processors with out-of-order execution, superscalar stages, pipelined functional units, etc. This is done within an automated work flow, presented in Figure 1.

4. THE SIM-NML LANGUAGE

[1]A basic block is a sequence of instructions, without any branch, which makes up the execution path of a program.

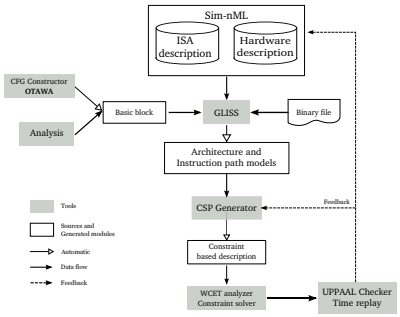

Figure 1: ADL-based work flow for WCET analysis

A representation of an architecture consists of the description of its hardware components and the supported instruction set. Sim-nML [11] is a hierarchical and a highly structured language able to perform such a description. From this, it provides the ability to generate processor specific tools. In Sim-nML, the processor model is described at instruction level, as a hierarchical structure using an attributed grammar. The instructions and the addressing modes are described by pre-defined attributes. The *syntax* attribute defines the assembly representation of the instruction. The attribute *image* gives the binary representation and the attribute *action* defines the semantics of the instruction. See lines 15-18 of listing 1.

Listing 1: Sim-nML processor description

```
1  stage FE , DE , IS , ALU[2] , MEM , CM
2  extend FE , DE , IS , CM
3      capacity = 2          // degree of super-
           scalarity
4      inorder = true        // in-order stages
5  extend ALU , MEM
6      inorder = false // out-of-order stages
7
8  // Fetch Buffer and Re-order Buffer
9  buffer FBuf [4] , RoB [8]
10
11 reg PC [1,card(32)] // 32-bit PC register
12 reg R [32,card(32)] // 32 registers of 32 bits
13 mem M [32,card(8)]// a memory of 2^32 8-bit words
14
15 op add (d:card (2),s1:card (2),s2:card (2))
16  syntax = format ("add r%d r%d r%d",d,s1,s2)
17  image = format ("00%2b%2b%2b",d,s1,s2)
18  action = {R[d] = R[s1] + R[s2] ;}
19  uses = FE & FBuf,DE, IS & RoB,ALU[0] & R[s1].read
        & R[r2].read & R[d].write & RoB #{1}, CM
20
21 op load (d:card (2),s:card (2))
22  ...
23   uses = FE & FBuf,DE,IS & RoB,MEM & R[d].write &
         R[s].read & M.read & RoB #{10}, CM
```

We extended the Sim-nML language such that we can declare the hardware structure of the processor[2]. Precisely, the extended language provides the syntax to define the pipeline stages, the resources accessed by the instructions when they execute on the pipeline. The properties of these hardware components are specified as attributes. So, we can declare stages, buffers, registers and memories as hardware resources. Lines 1-13 of Listing 1 describes the processor in the Figure 2. The instruction definition is extended with an attribute *uses* that describes the execution model of the instruction. The execution model represents the instruction behavior in terms of resources allocation. For example, to begin executing on a stage, an instruction has to wait for its resources to be available. Therefore, the execution time of an instruction is impacted by the general resources state. The *uses* attribute defines, in a timed sequence called clause, the resources used by an instruction in each step of its execution. A sequence is defined using commas. Every clause in a sequence represents a step of the instruction execution. In every step, one or more resources are required, and access can be in a read or a write mode. Parallel access is expressed by an operator &. Access to some resources can take a fixed duration t that can be specified as $\#\{t\}$. An example of *uses* attribute is given in lines 19 and 23 of Listing 1.

[2]Extensions are underlined in the listings.

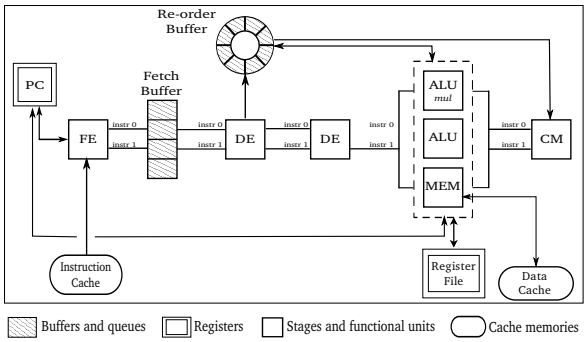

Figure 2: An out-of-order superscalar processor

Listing 2: Specialized execution with different latencies

```
1  stage FE , DE , IS , ALU[2] , MEM , CM
2  extend DIV
3   uses = FE , DE , IS , ALU[1] & R[rn].read &
4          R[rd].write & R[rm].read #{25} , CM
5  extend MUL
6   uses = FE , DE , IS , ALU[1] & R[rd].write &
7          R[rm].read & R[rn].read #{5} , CM
```

Listing 3: Multiple load instruction

```
1  extend load_multiple
2   uses = FE , DE , IS ,
3    (if reglist<0..0> == 1 then MEM & M.read & R
         [0].write .. endif),
4    (if reglist<1..1> == 1 then MEM & M.read & R
         [1].write .. endif), ... , CM
```

The language was extended to handle some complex features of recent architectures. In fact, modern architectures present complex pipelines and instructions with complex execution models. We were able to handle pipelines with specialized execution units, micro-coded instructions and pipelined functional units (like floating point pipelines). For example, we assume having a pipeline with 2 out-of-order ALU units, among which only one executes multiplication instructions. This feature is relevant in execution time computation. Multi-cycle instructions and the micro-coded instructions as the load/store multiple are handled by our description language. Listing 3 presents a load multiple instruction where the *reglist* parameter is a bit sequence. A

bit is set to one if the corresponding register is loaded. Different latencies can be specified for every stage and are taken into consideration when generating temporal constraints. The example of Listing 2 considers the same architecture in Figure 2. We assume that multiplication and division instructions are executed by the second *ALU* unit, which is the specialized functional unit. However, different latencies are specified for the two instructions on that functional unit (see Listing 2). These clauses, specified for every instruction supported by the processor, will be used, with the stages attributes to generate the instructions constraints [12]. In fact, we use temporal intervals to represent the lifetime of instructions on the pipeline stages and the resource allocations. The instruction dependencies within a basic block are expressed with constraints on the time intervals. The constraint description captures the architecture and instruction semantics such as the resource allocation strategy, the data dependencies, the structural dependencies, contentions on shared resources, etc. The constraints are combined to formulate a CSP, which resolution provides the time cost of basic blocks of a program.

5. VALIDATION AND ANIMATION

The aim of the OTAWA tool is to provide an environment for the hardware architect. In this section, we consider two tools related to validation and animation. These tools are based on an internal representation modeling the architecture and the instructions behavior.

ISA level tasks:	**Step level tasks:**
$\langle \text{ISA} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle}$	$\langle \text{Step} \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$
Basic tasks (leaves):	
$\langle \text{occurrence}\vert ? \rangle \langle Stage \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle}$ |	
$^{[r\vert w]}_{\langle \text{occurrence}\vert ? \rangle} \langle Resource \rangle^{\langle \text{interval} \rangle}_{\langle \text{instruction} \rangle, \langle \text{step} \rangle, \langle \text{index} \rangle}$	
$\langle Stage \rangle$, $\langle Resource \rangle ::= \langle Register \mid Buffer \mid Memory \rangle$	

Table 1: Collected clauses syntax

5.1 The internal representation

Since we are concerned by modeling concurrency of instructions, we have chosen an OCCAM based representation [8, 13]. We consider the following basic constructors:

- `USES` : This is the terminal case in which a basic resource request and an access are specified.
- `SEQ` : This is a sequential behavior. Each element of this sequence is specified recursively by a clause. Intuitively, the `SEQ` constructor will allow us to specify the execution path of an instruction. Each clause of this path will specify the local behavior with respect to a stage of the processor, what we called a step.
- `PAR`. This is a concurrent behavior. Each element is specified recursively by a clause. Intuitively, we express as such the simultaneous use of resources during a step.
- `ATTR`. These are general attributes superposed to a clause, e.g., timing ones. Actually, our internal representation is decorated with the resolved constraints.

Our internal representation is based on generic attributed clauses. For our validation purposes, we instantiate the attribute type as the corresponding time interval. With respect to our concerns, the collected clauses can be described through the light DSL (Domain Specific Language) given by the syntax in Table 1. For instance, if we consider the load instruction : `ldr r3 , [ r11 , -#20 ]` executed on the architecture of Listing 1, the following clause represents the effective resources access of the instruction.

$$i0\_clause = {}_0FE^{[0,1]}_{0,0} \ \& \ {}^{?}_{?}FBuf^{[0,1]}_{0,0} \ \& \ {}^{r}_{15}R^{[0,1]}_{0,0} \ , \ {}_0DE^{[1,2]}_{0,1} \ ,$$
$${}_0IS^{[2,3]}_{0,2} \ \& \ {}^{?}_{?}RoB^{[2,3]}_{0,2} \ , \ {}_0MEM^{[3,4]}_{0,3} \ \& \ {}^{r}_{0}M^{[3,4]}_{0,3} \ \& \ {}^{r}_{11}R^{[3,4]}_{0,3}$$
$$\& \ {}^{w}_{3}R^{[3,4]}_{0,3} \ \& \ {}^{?}_{?}RoB^{[3,4]}_{0,3} \ , \ {}_0CM^{[4,5]}_{0,4}$$

5.2 Validation.

We have studied two kinds of validation:

- *Instruction constraints validation.* It consists in checking that the results are coherent with respect to the initial representation. For example, we validate that the intervals of instruction steps respect the data hazard constraints.
- *Architecture validation.* It consists in checking that the results are coherent with respect to the studied architecture. For instance, we validate that, in an in-order pipeline, an instruction steps occurs before its successors.

$$\forall \ s^{[l,u)}_{i,s} \in STEP. \ \forall \ s^{[l',u')}_{i',s'} \in STEP. \ i < i' \Rightarrow u \leq l'$$

Although, theoretically these validations should not be necessary, our experiments have shown that they are of great help. Actually, it is much easier to assess these validations than those on the usual execution graphs [14] that can be huge.

5.3 Animation.

The aim of the "animated" views is to assist the architect to better understand instruction behaviors. For that purpose, we consider a well known formal model: that of timed automata [1] for which verification tools like UPPAAL [5] implement the decision procedure. As a matter of fact, the UPPAAL framework is by now a mature tool which offers a powerful simulator in order to interact dynamically with the architecture under study. Currently, we generate automatically the *instruction view* and the *stage view*. Also, architects can use the UPPAAL query language to express temporal predicates. Then, such predicates will be decided automatically. Moreover, if some property is not verified a counter example is exhibited. To summarize, the architect user can step along the execution of his model and validate general dynamic properties.

Behavior specification: from timed clauses to timed automata.

Basically, to each clause, we associate a timed automata location. Thanks to an invariant, control remains in such a location starting from the lower bound until the upper bound of the interval associated to the clause is reached. A guard ensures that such a location is not left before the upper bound is actually reached. Each of our animated views consists in a network of such automata sharing a global clock `clk`. Last, since our automata progresses according to time (not to internal events or synchronizations), the labels of their states are also meaningful.

- The instruction view. In this view, a timed automata was assigned to each instruction. Stepping trough this view allows us to see how each instruction evolves. This view is especially interesting for observing the relative "speed" of each instruction: when an instruction enters the pipe and maybe stalls over its successive stages.
- The stage view (Figure 3). In this view, to each stage is associated a timed automata. Stepping through this view allows us to see how stages evolve. This view is especially interesting for observing stages occupancy.

Figure 3: Stage view network timed automata

6. CONCLUSION

In this paper, starting from our extension of the OTAWA tool allowing WCET computations for today architectures [12], we have extended the Sim-nML language in order to handle modern processors features. We found the constraint-based time computation method suitable for expressing complex instruction features. We have also presented a light DSL for expressing architecture properties. Last, we have considered how to validate and animate the results obtained through constraint solvers. As a future work, we intend to use the DSL presented in this paper to formalize the architecture specification and constraints, in order to elaborate a reliable description of the architecture constraints.

7. REFERENCES

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Comput. Sci.*, 126(1):183–235, February 1994.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In *CPP*, pages 135–150, 2011.

[3] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2010.

[4] J.-L. Béchennec and F. Cassez. Computation of wcet using program slicing and real-time model-checking. *CoRR*, 2011.

[5] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST*, pages 125–126, 2006.

[6] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, Mar. 2007.

[7] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[8] A. Burns. *Programming in Occam 2*. Addison-Wesley, 1988.

[9] A. Dalsgaard, M. Olesen, M. Toft, R. Hansen, and K. Larsen. METAMOC: Modular execution time analysis using model checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[10] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. European Design and Test Conference (EDTC), 1995.

[11] M. Freericks. The nml machine description formalism. Technical Report 1991/15, TU Berlin, 1991.

[12] H. Herbegue, H. Cassé, M. Filali, and C. Rochange. Hardware architecture specification and constraint based wcet computation. In *International Symposium on Industrial Embedded Systems (SIES)*, June 2013.

[13] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 2006.

[15] P. Mishra and N. Dutt. Modeling and validation of pipeline specifications. *ACM Trans. Embed. Comput. Syst.*, pages 114–139, Feb. 2004.

[16] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer, 2002.

[17] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, 2000.

[18] C. Rochange and P. Sainrat. A context-parameterized model for static analysis of execution times. *Transactions on High-Performance Embedded Architectures and Compilers II*, 2009.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.